Protocolos de Pertenencia a Grupos para Entornos Dinámicos

Group Membership Protocols for Dynamic Environments

UNIVERSIDAD POLITÉCNICA DE VALENCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

TESIS DOCTORAL

Presentada por: Mª Carmen Bañuls Polo

Dirigida por: Pablo Galdámez Saiz

Diciembre 2005

A Emi

Gracias...

En primer lugar me gustaría dar las gracias a mi director de tesis, el Profesor Pablo Galdámez Saiz, por su apoyo y colaboración a lo largo del trabajo científico, y por su ayuda para lidiar con los trámites burocráticos en la etapa final de esta tesis.

Al Profesor José M. Bernabéu Aubán agradezco que me diera en primer lugar la oportunidad de entrar a formar parte del grupo de Sistemas Distribuidos. Asimismo, agradezco al Instituto Tecnológico de Informática el haberme permitido investigar durante estos años los problemas de algoritmos distribuidos tratados en esta tesis, y compaginar mi trabajo con la investigación y la docencia en otros campos.

Mi agradecimiento también es para el profesor Francesc D. Muñoz Escoí, cuya revisión y asesoramiento han sido de gran ayuda a lo largo de las distintas fases del trabajo. También agradezco al profesor Pietro Manzoni las sugerencias que llevaron a desarrollar la última parte de esta tesis.

Agradezco igualmente a los revisores de este manuscrito los comentarios y correcciones que han contribuido a mejorarlo y a darle su forma definitiva.

Quiero dar las gracias a todos los miembros de los grupos de investigación de Sistemas Fiables y Sistemas Distribuidos del ITI, y en particular a Emili Miedes y Juan Carlos García, con quienes he colaborado directamente en algunos de los últimos trabajos, y a Rubén de Juan por su colaboración en la revisión de este texto. Hago extensivo mi agradecimiento a todos los compañeros del Instituto por su ayuda con los diferentes problemas técnicos y por hacer el trabajo de cada día más ameno y gratificante.

Por último, aunque no por ello menos importante, agradezco a toda mi familia su apoyo y ánimo. Y muy especialmente agradezco a Emi el

que esté siempre a mi lado, confiando en mí y alentándome a continuar cuando no sé cómo seguir adelante.

Contents

Resumen

Los sistemas distribuidos gozan hoy de fundamental importancia entre los sistemas de información, debido a sus potenciales capacidades de tolerancia a fallos y escalabilidad, que permiten su adecuación a las aplicaciones actuales, crecientemente exigentes. Por otra parte, el desarrollo de aplicaciones distribuidas presenta también dificultades específicas, precisamente para poder ofrecer la escalabilidad, tolerancia a fallos y alta disponibilidad que constituyen sus ventajas. Por eso es de gran utilidad contar con componentes distribuidas específicamente diseñadas para proporcionar, a más bajo nivel, un conjunto de servicios bien definidos, sobre los cuales las aplicaciones de más alto nivel puedan construir su propia semántica más fácilmente.

Es el caso de los servicios orientados a grupos, de uso muy extendido por las aplicaciones distribuidas, a las que permiten abstraerse de los detalles de las comunicaciones. Tales servicios proporcionan primitivas básicas para la comunicación entre dos miembros del grupo o, sobre todo, las transmisiones de mensajes a todo el grupo, con garantías concretas. Un caso particular de servicio orientado a grupos lo constituyen los servicios de pertenencia a grupos, en los cuales se centra esta tesis. Los servicios de pertenencia a grupos proporcionan a sus usuarios una imagen del conjunto de procesos o máquinas del sistema que permanecen simultáneamente conectados y correctos. Es más, los diversos participantes reciben esta información con garantías concretas de consistencia. Así pues, los servicios de pertenencia constituyen una componente fundamental para el desarrollo de sistemas de comunicación a grupos y otras aplicaciones distribuidas.

El problema de pertenencia a grupos ha sido ampliamente tratado en la literatura tanto desde un punto de vista teórico como práctico, y existen múltiples realizaciones de servicios de pertenencia utilizables. A pesar de ello, la definición del problema no es única. Por el contrario, dependiendo del sistema concreto para el cual se diseñe el servicio, se requieren de este unas garantías u otras. Las especificaciones e implementaciones originales estaban enfocadas a escenarios *clásicos*, donde un número de nodos reducido se interconectaban a través de redes con relativamente pocos fallos y de comportamiento bien conocido (típicamente redes de área local). Sin embargo, con el desarrollo de los sistemas modernos, la expansión de Internet, la generalización de las comunicaciones inalámbricas, etc., se ha hecho predominante otro tipo de sistemas

distribuidos más dinámicos para los cuales los servicios distribuidos clásicos no resultan convenientes. Por ese motivo, se han desarrollado en los últimos años servicios de pertenencia a grupos a propósito para otro tipo de entornos.

En este trabajo nos hemos centrado en la especificación y realización de servicios de pertenencia a grupos para determinados sistemas dinámicos que presentan interés de cara al desarrollod e eplicaciones altamente disponibles. En concreto, hemos estudiado tres tipos de escenarios.

- Sistemas con topología bien conocida, y tamaño moderado, en los que las particiones son bastante probables, como podría ser el caso de un sistema compuesto por varios nodos o clusters localizados, conectados entre sí por enlaces WAN de alta velocidad. Para estos sistemas hemos definido e implementado el servicio de pertenencia a grupos particionable HMS, que proporciona las garantías de consistencia más estrictas de un servicio clásico de pertenencia, más una propiedad adicional. Esta propiedad, o *Acuerdo uniforme mayoritario* facilita la recuperacin de las aplicaciones usuarias tras la reparación de fallos y particiones, gracias al mantenimiento de la historia mayoritaria de forma uniforme dentro del grupo.

- Arquitecturas cliente–servidor a gran escala, en las cuales un reducido grupo de servidores atiende las peticiones de un gran conjunto de clientes dinámico y de identidades y tamaño no conocidos de antemano. Para este tipo de escenario hemos especificado un servicio de pertenencia de clientes, que proporciona a clientes y servidores la información más relevante respecto al grupo opuesto, para mantener las conexiones de los clientes al grupo de forma unificada y simplificar la reacción a fallos de cualquiera de los dos tipos de nodo. La especificación se ha realizado en la definición e implementación completa del protocolo HaloMS, también descrito en este trabajo.

- Las redes móviles ad hoc o espontáneas introducen nuevos retos para el desarrollo de servicios distribuidos. En concreto el consumo de energía y ancho de banda son en este caso parámetros a tener en cuenta a la hora de diseñar protocolos. Para este tipo de sistemas, y otros con similares requisitos de ahorro de recursos, hemos especificado un Servicio de Pertenencia a Grupos Bajo Demanda, capaz de proporcionar garantías de consistencia estricta solo durante periodos de tiempo finitos, que se deciden en función de las exigencias de las aplicaciones. Esto permite evitar el gasto innecesario de recursos energéticos y de comunicaciones derivado de rondas de acuerdo y costosas reconfiguraciones cuando ningún componente está haciendo uso de de la información de pertenencia. También presentamos aquí la realización de este servicio en el protocolo MODUS, que implementa la especificación Bajo Demanda a partir de un servicio de pertenencia clásico ya existente.

  Específicamente para el escenario *ad hoc*, además, la falta de una topología predefinida supone una dificultad añadida para el desarrollo de servicios de pertenencia, y de cualquier protocolo basado en acuerdo distribuido. Esta carencia de conocimiento inicial puede suplirse mediante un servicio de estimación de pertenencia que no proporcione consistencia estricta. Hemos analizado una propuesta de servicio de estimación de tipo epidémico, que sirve como base para el desarrollo de otros protocolos más restrictivos, de los cuales pertenencia a grupos estricta sería un ejemplo. En nuestro análisis hemos tenido en cuenta que tal servicio deberá ser utilizado conjuntamente con un servicio de encaminamiento. Basándonos en la simulación de los protocolos, hemos estudiado dos posibles arquitecturas que proporcionan ambos servicios y comparado su rendimiento en diferentes escenarios, en términos de su consumo energético y de la calidad de la información de pertenencia facilitada.

Este trabajo describe la especificación de todos estos servicios, y la implementación de los correspondientes protocolos. Todos estos han sido implementados completamente y están disponibles para su descarga en la web. Puesto que el objetivo del trabajo es ofrecer un soporte de uso práctico para el desarrollo de aplicaciones distribuidas en los diferentes escenarios estudiados, todas las implementaciones encajan en una arquitectura modular más general, caracterizada por la independencia de cada uno de los servicios, implementados como componentes autónomas. Estas componentes tienen asimismo interfaces bien definidas que pueden ser utilizadas indistintamente por otros servicios del sistema o por aplicaciones de alto nivel.

Resum

Els sistemes distribuïts gaudeixen hui en dia d'una importància fonamental dins dels sistemes d'informació, degut al seu potencial d'escalabilitat i tolerància a fallades, que permiteixen la seua adequació a les aplicacions actuals, cada vegada més exigents. D'altra banda, el desenvolupament d'aplicacions distribuïdes presenta també dificultats específiques, precisament per tal d'aconseguir l'escalabilitat, tolerància a fallades i alta disponibilitat que constitueixen els seus principals avantatges. Per això és de gran utilitat comptar amb components distribuïdes disenyades específicament per proporcionar, a un nivell més baix, un conjunt de serveis ben definits, per damunt dels quals les aplicacions de més alt nivell puguen construir la seua pròpia semàntica amb més facilitat.

És el cas dels serveis orientats a grups, d'ús molt comú per part de les aplicacions distribuïdes, a les quals permiteixen abstraure's dels detalls de les comunicacions. Aquests serveis ofereixen primitives bàsiques per a la comunicació entre els membres del grup i, principalment, per a la transmissió de missatges a tot el grup, amb garanties particulars. Un cas particular és el constituït pels serveis de pertinença a grups, en què aquesta tesi es centra. Els serveis de pertinença a grups proporcionen als seus usuaris una imatge del conjunt de processos o màquines del sistema que romanen connectats i correctes simultàniament. Més encara, els diversos participants reben aquesta informació amb garanties de consistència determinades. Així els serveis de pertinença constitueixen una component fonamental pel desenvolupament de sistemes de comunicació a grups i d'altres aplicacions distribuïdes.

El problema de pertinença a grups ha estat profusament tractat en la literatura, tant des d'un punt de vista teòric com pràctic, i existeixen multitud d'implementacions de serveis de pertinença utilitzables en la pràctica. Malgrat tot això, la definició del problema no és única. Pel contrari, depenent del sistema particular pel qual es disenye el servei, s'exigeixen d'ell unes garanties o altres. Les especificacions i implementacions originals estaven destinades a escenaris *clàssics*, als quals un nombre reduït de nodes es connectaven mitjançant xarxes de comportament ben conegut amb fallades relativament escasses (típicament xarxes d'àrea local). Tanmateix, amb el desenvolupament dels sistemes moderns, l'expansió d'Internet, la generalització de les comunicacions sense fils, etc., altres tipus de sistemes distribuïts més dinàmics s'han convertit en

predominants. Per a ells els protocols clàssics no resulten ja convenients. Per això els últims anys s'han desenvolupat serveis de pertinença a grups expressament per diversos tipus d'entorns.

En aquest treball ens hem centrat en l'especificació i realització de serveis de pertinença a grups per una sèrie determinada de sistemes dinàmics que presenten interès pel desenvolupament d'aplicacions altament disponibles. En particular, hem estudiat tres tipus d'escenari.

- Sistemes amb topologia coneguda i tamany moderat, als quals les particions són relativament probables, com podria ser el cas d'un sistema format per diversos nodes o clusters localitzats, interconnectats per enllaços WAN d'alta velocitat. Per aquests sistemes hem definit i implementat el servei de pertinença a grups particionable HMS, el qual proporciona les garanties de consistència més estrictes d'un servei de pertinença clàssic, a més d'una propietat adicional. Aquesta propietat, anomenada *Acord uniform majoritari*, facilita la recuperació de les aplicacions després que una fallada o partició ha sigut reparada, gràcies al manteniment de la història majoritària de forma uniforme dins del grup.

- Arquitectures client–servidor a gran escala, a les quals un grup reduït de servidors atén les peticions d'un ample conjunt de clients, la identitat i el nombre dels quals no són coneguts per endavant. Per aquest tipus d'escenari hem especificat un Servei de Pertinença per Clients, que proporciona a clients i servidors la informació més relevant respecte a l'altre grup, per tal de mantenir les conexions dels clients al grup de manera unificada i simplificar la reacció a fallades d'ambdós tipus de node. L'especificació s'ha realitzat en la pràctica amb la definició i implementació completa del protocol HaloMS, també descrit en aquest treball.

- Les xarxes mòbils *ad hoc* o espontànies introdueixen nous reptes pel desenvolupament de serveis distribuïts. En particular, el consum d'energia i d'amplària de banda esdevenen paràmetres rellevants a l'hora de disenyar protocols. Per a aquest tipus de sistema i d'altres amb similars requisits d'estalvi de recursos, hem especificat un Servei de Pertinença a Grups Sota Demanda, capaç d'oferir garanties de consistència estricta sols durant períodes finits de temps, decidits en funció de les exigències d'informació de pertinença per part de les aplicacions. Això permet evitar el consum innecessari de recursos energètics i de comunicacions que es derivaria de rondes d'acord i costoses reconfiguracions quan ningú no està fent ús de la informació de pertinença. Així mateix presentem ací la realització d'eixe servei en el protocol MODUS, el qual implementa l'especificació Sota Demanda a partir d'un servei de pertinença clàssic ja existent.

  Específicament per l'escenari *ad hoc*, a més, la falta d'una topologia predefinida suposa una dificultat afegida pel desenvolupament de serveis de pertinença, i de qualsevol altre protocol distribuït basat en acord. Aquesta manca de coneixement inicial pot suplir-se amb un servei d'estimació de pertinença que no pretenga proporcionar consistència estricta. Hem analitzat una proposta de servei d'estimació de tipus epidèmic, que serveix com a base pel desenvolupament d'altres protocols més restrictius, dels quals el de pertinença a grups seria un exemple. La nostra anàlisi ha tingut en compte que aquest servei haura d'ésser utilitzat conjuntament amb un servei bàsic d'encaminament. Basant-nos en la simulació de protocols, hem estudiat dues possibles arquitectures que proporcionen ambdós serveis i hem comparat el seu rendiment en diversos escenaris, en termes de consum energètic i de qualitat de l'estimació de pertinença.

Aquest treball describeix l'especificació de tots els serveis esmentats, i la implementació dels corresponents protocols. Tots ells han sigut completament implementats i estan disponibles per la seua descàrrega a la web. Puix que l'objectiu del treball és oferir un suport utilitzable en la pràctica pel desenvolupament d'aplicacions distribuïdes als diferents escenaris descrits, totes les implementacions s'ajusten a una arquitectura modular més general, caracteritzada per la independència de cadascú dels seus serveis, els quals són implementats com a components autònomes. Tals components tenen també interfícies ben definides que poden ser utilitzades indistintament per altres serveis del sistema o per les aplicacions d'alt nivell.

Abstract

The importance of distributed systems in current information systems is due to their potential capabilities to support the growing requirements of increasingly demanding applications. However, developing distributed applications is not easy. On the contrary, specific difficulties arise as they are devised to offer scalability, fault tolerance and higher availability. Therefore, it is most useful to count on specific distributed components that provide, at a lower level, a suit of well–defined services upon which higher level applications, or other such services, can build up.

This is the case of group oriented services, extensively used by distributed applications, as they allow them to abstract from the details of communications. Such services provide basic primitives to achieve one-to-one and one-to-many communications with specific guarantees. One particular case of them are group membership services, on which this thesis is focused. Group membership services provide their users with an image of the set of system processes or machines that are simultaneously correct and connected. Moreover, the various participants receive this information with well–defined consistency guarantees. Thus they are a fundamental building block for group communication services, also useful for other distributed applications.

Although the group membership problem has been largely discussed in the literature from a theoretical and a practical point of view, and despite the existence of multiple empirical realisations, there is no unique formulation of the problem. On the contrary, the characteristics required from a membership service depend on the particular system for which it is devised. The original specifications and implementations were directed to classical settings, with reduced number of members, and well behaved connections (typically over LANs). However, with the development of modern systems, the expansion of the Internet, wireless communications, etc., more dynamic scenarios have become predominant. For them, the classical distributed services are not well suited. Different membership services have therefore then been defined for more changing environments.

In this work we focus on the specification and realisation of group membership services for certain dynamic systems of practical interest for highly available applications. In particular, we have studied three different scenarios.

- Systems with well–known topology and moderate size, where partitions are likely to occur, as could be the case of a system composed of several sites connected through a WAN with high speed links. For this kind of systems, we have designed and implemented a partitionable group membership service, HMS, that provides the most strict classical guarantees plus the property of *Uniform Majority View Agreement* to ease the recovery of user applications after a failed member rejoins the group.

- Large client–server architectures, where a reduced group of servers receive requests of a large, dynamic set of clients, of a priori unknown identities and number. For them a Client Membership Service has been specified that provides clients and servers with the relevant information regarding the opposite group, so to unify the maintenance of volatile client connections, also in case of core node faults. This specification has been realised and fully implemented in the HaloMS protocol, also described in this work.

- Mobile ad hoc networks introduce new challenges to the development of distributed services. In particular, the energy and bandwidth consumption become relevant for the design of protocols. For these and other systems with a similar concern for saving resources we have specified an On Demand Membership Service, capable of providing strong consistency guarantees only for finite periods of time, decided as a function of user application requirements. This avoids the resource consumption derived from agreement rounds and costly reconfigurations when no component is making use of membership information. We present also a realisation of this service in the MODUS protocol, which provides the On Demand guarantees from a generic membership service.

  For the specific scenario of ad hoc networks, in addition, the lack of a priori knowledge about the topology raises a special difficulty for the development of membership and other agreement based services. To supply this initial information we have analysed the usage of a membership estimation service in the epidemic style. Such service will not provide strict consistency guarantees but will serve as basis for more demanding protocols, e.g. strict group membership. We have taken into account that such service will be used in conjunction with routing. Our analysis, based on simulation, studies two possible architectures that provide both support services, and compares their performance in different scenarios.

This work describes the specification of all these services, and the implementation of the corresponding protocols. All of them have been fully implemented and are now available for downloading. Since the goal of this work is to offer a practical support for the development of applications in the various studied scenarios, all these implementations fit within a more general, modular framework. The latter is characterised by independent components for each service. These components have clearly defined interfaces that can be employed by user applications as well as by other services in the system.

Chapter 1

Introduction

1.1 Motivation and Scope

Nowadays the increasing requirements on systems and applications often exceed the capabilities of available single computers. Moreover, the usage of communication systems connecting computers at various levels is becoming a general practice, responding to the present trend towards globalisation. Distributed systems are thus becoming the general model, as distributed applications are extensively used.

The concept of distributed system applies to very different settings, from the widely used client–server architectures to computer clusters, including more recent scenarios as P2P systems and Grids. In all cases, they offer a number of advantages with respect to centralised systems, as their potential performance, scalability, reliability, etc. But they also present specific difficulties regarding their design and implementation. The coordination of distributed components often requires information on the state of the system, or particular guarantees related to communications and other fundamental services. Therefore distributed services that provide well–defined semantics may act as fundamental blocks to ease the building of correct distributed applications.

Some of the most significant among such services are the group oriented ones, widely used by highly available applications. These services help the development of distributed components by allowing the abstraction from the details of group communication and by providing guarantees upon which to build up the desired semantics. This thesis is focused on group membership services, one type of such fundamental distributed components, which can provide support both to other group communication components and to user level applications. The group membership problem has been extensively discussed in the literature, both from a theoretical [1, 2, 3, 4, 5] and from a practical point of view (e.g. [6, 7, 8, 9, 10, 11]). A unique specification of the problem does not exist but, informally speaking, a group membership service must provide its user applications or components with consistent information about the correct and failed processors in the system.

The variety of scenarios falling under the category of distributed systems implies also variated requirements on the support services that potential applications exploit. The firstly devised distributed systems were typically characterised by a reduced number of members and stable connections over, for instance, a LAN. Therefore the original distributed services, in particular group membership ones as [8, 9], were specified and designed for such scenarios. The cost of the strong consistency provided by *classical* algorithms may become excessive when the same algorithms are applied to larger scale or rapidly changing systems. Thus, with the utilisation of larger systems and wide area networks, alternative services (e.g. [7, 12, 13, 14]) were specially designed to adapt to the new system features. Some of them (e.g. [15, 16, 17]) implement different semantics, with more relaxed consistency guarantees.

The divergence between classical and more modern distributed systems becomes more evident the larger the scale and the faster the group changes. Our main interest is focused on the following particular scenarios, characterised by their dynamism and by their interest as frameworks for the deployment of distributed applications.

- Systems of moderate size and well–known topology where partitions are likely to occur. For instance, a system supporting a replicated service over different sites which are connected through a WAN over high speed links. In the following we will refer to these systems as WAN–wide clusters (or WAN clusters). Contrary to classical cluster–like environments, where network partitions are rare, in such a scenario it is possible that some sites become isolated. This scenario happens in systems as MADIS [18] and DeDiSys [19].

- Large scale client–server architectures, consisting of a reduced, preconfigured group of nodes serving requests from a set of external clients whose identities and number are not predetermined, and which will typically connect over a WAN. From a practical point of view, this is the typical scenario that appears on the interaction between clusters devoted to offer highly available services and clients that connect to access to those services.

  This problem involves the interaction between two node sets with very different properties, since the scale and changing rate of the group of clients is much higher than that of servers.

- Mobile ad hoc networks. They are entirely formed by wireless nodes, connected through a network that has no fixed structure. Failures and other changes are common to these systems, conferring them high dynamism. Moreover, they have other characteristic features, in particular the limited energy supply. Thus energy and bandwidth consumption become factors to be minimised by any devised protocol.

For each such system we have defined the guarantees to be provided by a usable membership service, ensuring their implementability by the definition of specialised protocols. Since the objective of this work is mainly to ease the development of higher level distributed components on such systems, all our protocols fit within the framework of a modular architecture in which the various services are clearly delimited and communicate only through well–defined, system independent interfaces.

1.2 Contributions

Partitionable group membership with uniform majority agreement. HMS is a membership service defined to fulfil the specification of the partitionable GMP and to be used as a membership service for WAN clusters. Although multiple specifications and implementations of GMP exist for such scenarios, HMS presents the particularity of being completely independent of communication guarantees, so that it is easy to couple it with any desired protocol, of whichever type, including diverse reliable or unreliable transport services and group communication components. As distinguishing features, HMS presents the particularity of clearly specifying the different treatment to minority and majority groups, and, most of all, it provides higher level components with a uniformity property regarding the majority group history.

While in a *primary component* membership service there exists a unique history of the group, in partitionable services disjoint views of the group may coexist, so that the concept of a unique group history disappears. Nevertheless, the history of majority groups *is* unique. Many applications making use of partitionable membership services need to be aware of the majority or minority character of their current view, in order to decide about their progress. When majority is lost and later recovered, special recovery protocols are typically run in order to resume their normal operation. The last property guarantees that HMS user applications can receive information about the past majority history and simplify such recovery.

The correctness of HMS with respect to the GMP specification has been shown and the protocol has also been completely implemented. We have produced a fully usable version that is available for download from the web [20].

Client membership service for large–scale systems. For client–server architectures we have specified a client membership service which provides servers with the most useful information about clients connections and disconnections. This is mainly intended for replicated services in which the servers form a group with the consistency guarantees of the *classical* GMP and have to handle possibly redundant invocations from a large and quickly changing group of clients. The client membership information is maintained consistently among the different servers that form a *classical* group, so that new clients are allowed to join the group and are assigned a unique identifier in an *open group* fashion.

Conversely the service provides clients with information about reachable server nodes, guaranteeing validity and a certain level of liveness of this information. Our specification is a realisation of the suggestion by Babaoglu and Schiper [21] about using different roles for group members in order to tackle the large scale problem.

The specification has been implemented by HaloMS, a client membership service which ensures the desired semantics with the support of a running membership service with the classical guarantees. HaloMS maintains client membership information tolerating server faults. It has been completely implemented in the framework of our modular architecture, and it has been tested using as support the implementation of HMS. A version is available for download from the web [20].

On demand membership service specification. For the particular case of bandwidth and energy aware networks we have proposed an On Demand Membership Service that provides the strong *classical* semantics of GMP only when required. This has the advantage of saving extra agreement rounds when no application or component is actually making use of the membership information, or does not need the consistency guarantees associated with the GMP. On the other hand, these same guarantees are indeed provided when an application requires them. Such a service is useful for the development of distributed services that need more than probabilistic QoS in the context of ad hoc networks.

MODUS is a membership service that implements the On Demand specification but making use of an existing classical membership service. We have defined the MODUS protocol and fully implemented it [20]. To evaluate its performance, it has been tested using again HMS as basis, and its performance has been evaluated with the message sending rate as metrics. Even not being the most efficient alternative to implement the on demand specification in an ad hoc network, MODUS still shows advantages with respect to a classic GMP approach and also when compared to a trivial On Demand realisation.

Membership estimation service for ad hoc networks. Finally, for the development of a membership service specific for ad hoc networks, we have proposed an alternative architecture. There, a membership estimation service based on gossip–style failure detection is used as support for a membership agreement component, which would use the inconsistent information of the former to launch agreement rounds when required. Nevertheless, if the goal is to optimise energy consumption while maintaining a certain QoS, it is not possible to separate the performance of such membership estimation from other necessary underlying services, as routing. Therefore, we have carried on a comparison of two different approaches that would provide both routing and estimation services, and we have analysed a set of scenarios in order to characterise their performance in terms of power consumption and liveness of the provided estimation. This type of combined study has not been performed before, to our knowledge, but makes clear that different services are inseparable if the performance of the whole system is to be optimised in terms of energy consumption.

1.3 Structure

The rest of this document is structured as follows.

Chapter 2 presents the preliminary material, including a general introduction to the group membership problem and a review of some of the most notable specifications and realisations of membership services for various types of systems.

Chapter 3 presents the basis for this work. From the architectural point of view, a modular framework is presented in which all our services fit. The general interfaces that regulate the interaction among the most relevant services are also introduced. From a more formal point of view, this chapter explains our choice for a stand-alone specification of the basic membership problem, decoupled from group communication properties. This is achieved by the set of safety and liveness properties extracted from [22, 10], whose enunciate is also presented in this chapter.

Chapter 4 describes the HMS protocol. HMS is a partitionable group membership service suitable for clusters or for a group of sites that have to provide distributed services over a WAN, but are interconnected with high speed links. Its operation and performance are detailed in this chapter, and its particularities are discussed. A more detailed proof of the algorithm correctness is provided in Appendix A.

In chapter 5 we present the specification of the client membership service and its implementation by the HaloMS protocol. As for the previous protocol, the correctness of the algorithm is proofed in Appendix A.

Chapter 6 introduces the specific features of ad hoc networks, that have motivated the particular on demand specification. The On Demand membership service specification is presented, and some alternatives for its implementation are discussed, including the detailed description of the MODUS protocol and the immediate alternative that achieves the on demand semantics from a basic group membership service. Different measurements were made to characterise their performances.

In chapter 7 we study a basic service for providing a membership estimation in an ad hoc network. We have studied how providing this and the routing service in a joint manner would affect the performance of the system, in terms of the relevant parameters of energy consumption and liveness of membership information. Finally, chapter 8 summarises our conclusions from this work.

Chapter 2

The Group Membership Problem

Informally speaking, the membership problem consists in the agreement among a group of processes about the composition of the group, i.e. the set of correct or operational processes. This information is provided to the group members in the form of *views*, which report in an ordered sequence those changes occurred to the group along the execution, including joined, failed and left nodes. The semantics provided by such a service is fundamental for the development of other distributed components.

Nevertheless, from a formal point of view, the specification of the problem is not unique. On the contrary, multiple definitions and implementations exist, depending on the type of system for which they are devised.

In this chapter, we try to locate the group membership problem in the context of distributed systems, highlighting its relationship with other fundamental problems in the area. Then, we review some of the most notable specifications and implementations of membership services for various system models, including the most static and reduced systems for which the original membership services were designed and the more dynamical settings that better fit the current distributed systems and on which our work is mainly focused.

2.1 The Group Membership Problem in Distributed Systems

Group membership services are fundamental components for fault–tolerance in distributed systems. Boosted by the development of practical solutions for group communication, a large number of membership services have been designed and used in the framework of various systems. The specification of the problem, however, has been subject to intensive research effort and discussion, with variated proposals for the formal enunciate of the problem and for experimental developments. In [4], Anceaume et al. evidenced the existence of flaws in the formalism of various specifications. Contrary to other well–known problems in the area of distributed systems, then, this one lacks a unique and well–defined formalisation. Moreover, the various specifications and existing protocols make use of different formalisms and assumptions on the system, which makes the comparison among them difficult.

Two types of membership services can be distinguished, namely *primary component* and *partitionable*. The former attempts to maintain a single image of the group composition, whereas the latter allows various views to exist concurrently in the group, so to reflect network partitions. *Partitionable* services, then, may deliver different (disjoint) images of the group membership to members of the different partitions.

The first specification of the primary–component group membership problem is due to Cristian [1] in the model of synchronous distributed systems. In this model, processes are equipped with synchronised clocks, and any message sent by a correct node is received by another correct node within a known delay.

In the asynchronous system model neither a global clock nor bounded delays exist. Given the impossibility to distinguish crashed nodes from very slow ones or large delays in network channels, the asynchronous model renders agreement problems much harder to solve. Furthermore, the original specification of the membership problem does not apply in such a system.

The first specification of a primary–component membership problem in an asynchronous model was done by Ricciardi and Birman [3, 8]. They assumed FIFO reliable communication channels and an unlimited space of potential members. Furthermore, the assumed failure model was crash/no recovery. The so called Strong Group Membership Problem was specified as a set of properties expressed as predicates on consistent cuts. Flaws of this specification were noticed in [4], as the incompleteness of predicate definitions, unreasonable requirements and failure to capture the intended semantics of primary–component.

A partitionable group membership problem was first presented by Dolev et al. [23, 6]. The crash/recovery failure model was assumed, and the problem was defined over a finite space of processes. All communication among processes was assumed to take place by means of reliable, causally ordered multicast. Failure detection was assumed to be done by the communication layer. The joint specification of membership and communication guarantees included virtual synchrony as one of the properties to be satisfied. In [4], however, it was noticed that the specification, even in its later refined version, also suffered from defects, as it allowed useless solutions in which the group was arbitrarily split for most of the time. Moreover, it permitted arbitrary removal of processes but did not require explicit actions after a suspect was received.

A different partitionable specification was due to Ezhilchelvan et al. [5]. It assumed communication among processes took place only by multicast addressed to the members of the group, over FIFO reliable channels. Although the failure model was crash/recovery, processes were not allowed to rejoin a group after leaving it, but were forced to start a new group. The specification also enclosed that of group communication guarantees, and was thus expressed as a set of properties regarding view consistency and message delivery, and including virtual synchrony and causally consistent total ordering of messages. As noticed in [4], however, the specification allowed a trivial solution where every member installed a singleton view.

Other authors have later proposed different specifications for practical membership services (for instance [7, 9, 10, 11]), but still there is none commonly accepted. More recently, the survey of group communication systems by Chockler et al. [22] identified a series of basic properties to be satisfied by every membership service and group communication protocol, and a set of optional properties that finally define the provided guarantees of each service. It also classified an important number of existing services according to their fulfilling of these properties.

The initial specifications were mainly focused on systems with a reduced number of members, relatively well connected (typically over a LAN). With the expansion of Internet, wireless technologies, etc., more dynamical scenarios became susceptible of hosting distributed applications and interest arose about membership services for such systems. Since applications for such environments may be quite specific and have particular requirements, different specifications have also appeared for special membership services that serve particular cases, broadening even more the diversity of specifications.

2.1.1 Consensus and Impossibility

As other agreement–like problems, the group membership problem (at least in its most traditional enunciates) can be reduced to consensus. It is then possible to build membership services from a basic consensus layer [7, 24].

Consensus is a basic problem in distributed systems [25]. The problem is defined for a finite set of processes. Each of them proposes an initial value and all the correct ones have to agree on a final value that corresponds to some proposal. Its formal specification consists of three properties.

Property C.1 (Termination)**.** *Every correct process eventually decides.*

Property C.2 (Agreement)**.** *Two correct processes do not decide differently.*

Property C.3 (Validity)**.** *If a process decides a value v, this was proposed by some process.*

Some of the most fundamental theoretical results in distributed systems are related to the consensus problem. In 1985, Fischer, Lynch and Paterson [26] proved that the consensus problem is not solvable by any deterministic algorithm in asynchronous systems where even a single process may crash. It is possible to circumvent such impossibility by relaxing the assumptions on the system, as considering partially synchronous models [27], or by equipping the system with a failure detector [28].

The impossibility result for consensus naturally maps in an impossibility result for membership and other consensus based problems. Chandra, Hadzilacos, Toueg and Charron-Bost [2] proved that it is also impossible to solve a primary component group membership problem in an asynchronous system where processes may fail. Their result does not hold however for partitionable specifications, which do not require that all correct processes agree on the current view. Nevertheless, partitionable specifications are at risk of being trivially solvable by arbitrarily splitting the group in singleton sets.

2.1.2 Failure Detection

The concept of failure detector was introduced by Chandra and Toueg [28] to overcome the impossibility of consensus in asynchronous systems. An unreliable failure detector provides each process with a list of other processes it currently suspects to have failed. These suspicions, however, may be incorrect and be later removed.

A failure detector is characterised by its completeness (it suspects all failed process) and accuracy (it does not suspect correct processes). Formally, the completeness property is enunciated as follows.

Property FD.1 (Strong Completeness)**.** *Eventually, every failed processes is suspected by every correct process.*

The variable degree of accuracy is formalised as the following four properties.

Property FD.2 (Strong Accuracy)**.** *No process is suspected before it crashes.*

Property FD.3 (Weak Accuracy)**.** *Some correct process is never suspected.*

Property FD.4 (Eventually Strong Accuracy)**.** *There is a time after which correct processes are not suspected by any correct process.*

Property FD.5 (Eventually Weak Accuracy)**.** *There is a time after which some correct process is never suspected by any correct process.*

Each one of these four properties, satisfied in conjunction with FD.1, defines one of the four independent classes of failure detectors in [28], respectively $\mathcal{P}$ (perfect), $\mathcal{S}$ (strong), $\diamond\mathcal{P}$ (eventually perfect) and $\diamond\mathcal{S}$ (eventually strong).

In [29], Chandra, Hadzilacos and Toueg proved that $\diamond\mathcal{W}$, a class of failure detectors equivalent to $\diamond\mathcal{S}$, is the weakest failure detector that enables consensus to be solved in asynchronous systems with a majority of correct processes.

Although eventual accuracy properties FD.4, FD.5 are not achievable in an asynchronous system, it is easy to build a failure detector that suspects all failed processes and thus satisfies strong completeness FD.1, for instance by means of periodic heartbeats and timeouts. Wrong suspicions cannot be completely eliminated, but in practice it is possible to reduce them so that the eventual accuracy properties hold for long enough[1] and the failure detector behaves as $\diamond\mathcal{P}$ or $\diamond\mathcal{S}$. For instance, in a heartbeat–timeout scheme, it is enough to enlarge the allowed timeout for a given process after a suspicion proves to be wrong, to virtually obtain an accurate failure detection.

Many specifications for the group membership problem thus consider a system model enriched with an eventually perfect failure detector.

2.1.3 Notation

Here we briefly introduce some terms widely used in the context of group membership services specification and description.

- View. Each one of the group images produced by a membership service is called a view, and represents a picture of the system status regarding failures and partitions. A membership service produces its output in the form of a succession of views that are delivered to the participants.

- Installation. The event of delivering a new view to a process is called installation. It is an irrevocable decision. In the remaining of this work we will sometimes use the terms *confirming* or *committing* to refer to the installation of a view, as a remainder of this irrevocability. In particular, we say a view is *committed* when it has been installed by some node.

- Virtual Synchrony. The concept of virtual synchrony was first introduced by Birman and Joseph [30]. It is the most popular property of group communication systems, and imposes that view change events are delivered in a consistent order with respect to the flow of messages. If two processes transit together from one view to the following, the virtual synchrony property requires that both of them deliver the same set of messages in the first view. This property gave origin to the virtual synchrony model, useful for distributed applications that implement replication. Other related models include Weak Virtual Synchrony [31], Extended Virtual Synchrony [32] and Optimistic Virtual Synchrony [33].

[1]Although formally wrong suspicions are required to *never* happen, it is enough that the accuracy property holds for long enough so that the algorithm can make progress.

2.2 Related Work: Group Membership Services

As discussed above, multiple practical solutions exist for group membership problems, that have been employed in a number of systems with variated properties and requirements.

Here we review some of the most significant, for either their usage in well–known systems or their tackling environments with particular features or necessities.

2.2.1 Classical Systems

The best known membership services are those used in the first and most popular distributed systems. Most of them were, at least originally, designed for reduced groups of computers, connected over a LAN.

Isis S-GMP. The first and best known group communication system, Isis, implemented as membership service a protocol that solved the Strong Group Membership Problem specified by Ricciardi and Birman [8]. S-GMP is a primary component service whose main goal is to provide a consistent and unique global view of the system. The service assumes reliable FIFO channels between every pair of processes, and the existence of a local failure suspecter that fulfils the strong completeness property FD.1.

The group is dynamic, with no predefined set of nodes. The algorithm uses, nevertheless, collective startup from a reduced and preconfigured set of members. Since the model of crash/no recovery is assumed, any node that recovers after a failure will use a different identifier and be treated as a new member by the protocol.

In the asymmetric, centralised protocol that solves S-GMP, one node plays the manager role. To exclude a failed node, the manager leads a two–phase protocol, in which it proposes the change to all members, and after receiving acknowledgements from all of them, commits the view change. When a new member wants to join, the manager directly submits the commit instruction for the enlarged group to the members of the view, after transferring the proper group state to the joined process. If the manager itself is faulty, a three–phase algorithm is run so to recover agreement. The node that initiates the reconfiguration must first interrogate the others about their local view and any submitted update not yet committed by the old manager. After collecting the answers, the initiator decides what is the update to be submitted.

Although Isis' original membership service was not partitionable, its successor, Horus [34], employed a variation of this algorithm in which partitions were allowed to occur. Nevertheless, a majority group was supposed to exist, so that if the group split in minority groups, the majority could not be recovered.

Transis. The membership service designed for Transis communication system [35] was the first partitionable one. The system assumes a finite set of processes, which communicate by asynchronous multicast, so that every message is broadcast to the whole universe of machines. The failure model is crash/recovery, and the network can partition and remerge. In absence of partitions, the transport is assumed to be reliable. The membership service operates on top of the communication layer, which provides causally ordered multicast and notification of failure suspicions, behaving in practice as a $\diamond\mathcal{S}$ failure detector.

Every node starts in a membership singleton containing only itself. The specification includes virtual synchrony, thus the membership algorithm takes charge also of delivering regular messages to the application level, relatively ordered with respect to views.

The protocol is completely symmetric. When a node suspects a failure or it detects the presence of a new node, it broadcasts a message containing its current idea of failed and live nodes. All nodes receiving this message merge the information with their own and broadcast their own knowledge about failures and correct nodes. When an identical message has been received from all the nodes that are believed to be alive, the membership view can be delivered to the application, after any causally preceding message. If failures occur during this exchange of messages, agreement on the proposed view may be impossible. In that case, if the surviving nodes had broadcast identical information, the view is notified as a failed intermediate proposal when a subsequent agreement is reached.

Totem. Totem [9] is a partitionable group communication system which provides totally ordered broadcast in a LAN. The membership protocol is based on that of Transis, described above, and is also partitionable. Totem constructs a logical ring along which a token circulates. Total order is ensured by only allowing the token holder to broadcast messages, with a monotonically increasing sequence number carried by the token.

The system consists of a finite set of processes, each of them with a unique identifier and some stable storage. Different to the protocols above, the underlying transport is not assumed to be reliable, and retransmission of messages under request is thus part of Totem protocols.

The membership protocol is invoked whenever a failure, partition, or token loss is detected or when a message from a foreign process is received. This protocol, similar to the former one, runs a *gather* phase in which messages containing the estimated failed and live sets are exchanged by all processes. When identical sets are received from all live members, a new ring is established, and an elected representative reconstructs and sends the token, which carries the commit order for the members of the formed configuration.

The guarantees provided by Totem include extended virtual synchrony, so that when a view change occurs, before the agreed view a transitional configuration is delivered to surviving nodes, to allow delivery of their pending messages, followed by the definitive view.

A more recent variant of Totem [36] employed multiple rings to guarantee totally ordered broadcast over a number of LANs interconnected by gateways. The membership of each individual LAN is maintained by the single–ring protocol, which triggers the higher level topology maintenance when configuration changes. Gateways are in charge of forwarding messages across local networks, and also of transmitting configuration changes while ensuring the total order of delivered messages.

Relacs. The Relacs project [37] specified the membership and multicast services for partitionable asynchronous environments. Remarkably enough, the specification of both services was completely decoupled.

The system is composed of a fixed set of processes holding unique and invariant identifiers. The assumed failure model is crash/no recovery. A failure detector that behaves as the partitionable version of $\diamond\mathcal{P}$ is assumed. The system is also assumed to satisfy an eventual symmetry property of the reachability relationship. The communication layer implements a reliable multisend primitive, and guarantees FIFO order. It also notifies suspects to the membership service.

The membership service, PGMS [10], is asymmetric. A node starts in a singleton view containing only itself. When a change occurs, the protocol is activated. Then a synchronisation phase takes place in which every process sends a sequence number to the set of members it believes to be reachable and waits until it has an answer from every non–suspected member. Then the estimation exchange is initiated, in which every process multisends its own estimate to all the members as many times as the composition varies. A coordinator is also elected and after observing agreement in the estimate messages, it sends a message with the new identifier and composition. Upon its reception some members may not install the complete view but only a *partial view* that ensures members come from the same or disjoint views.

Once the protocol is running the agreement phase, the view estimate cannot add new members, so that if a join occurs, the protocol will have to be run again after completing.

2.2.2 Large Scale Systems, WAN Networks

With the development of network technology and the availability of lower cost computers, distributed computations over large scale systems have acquired growing importance. The distributed protocols that support coordination and consistency have to satisfy new requirements as distributed applications have to be deployed in more demanding scenarios. If the original distributed systems were designed for small number of nodes, interconnected over a local area network, nowadays systems have to run on more dynamical settings, typically involving connections over WANs and much large number of nodes.

The increasing size of distributed systems makes scalability a central aspect of the design of distributed services. On the other hand, wide area networks also pose some specific challenges to the deployment of distributed protocols.

- High loss rates and latency, as compared to LANs. This works against the implementation of efficient reliability mechanisms, much easier to achieve on a local area network.
- Unpredictability of such characteristic parameters, as they may greatly vary across the network.
- Dynamism of the composition of the system, as connectivity changes may occur with relatively high probability, and lead to frequent failure suspicions.

A high number of attempts have been made to design specific protocols for these environments in the area of group communication, including membership services, facing one or more of their specific problems.

Phoenix. The goal of Phoenix environment [7] was to implement the virtual synchrony abstraction over a large, partitionable area network.

The system consists of a finite number of nodes, and the failure model is crash/recovery, but after a failure each node uses a different incarnation number. Partitions are allowed to occur, and a primary partition is ensured to exist and progress, whenever that is theoretically possible. The communication is assumed to be reliable and a local failure suspecter exists at every node. Suspicions may initially be revoked, if evidence of their incorrectness is received, but become stable at the second phase of the algorithm. The properties of the failure detector must be those of $\diamond\mathcal{W}$ for the protocol to achieve progress.

Phoenix membership service is integrated with the virtual synchrony guarantees. The problem is faced as an agreement problem about the membership of the group and the set of messages to be delivered. The solution is therefore based on consensus. The protocol has two phases. During the first one, an estimation is calculated to be used as input for the second one, in which a consensus protocol, based on that of Chandra and Toueg [28], is run.

When one or more processes are suspected, every member multicasts its set of delivered messages in the previous view and waits until it has received the corresponding set from every non–suspected member. At that point, the union of the received sets and the group of non–suspected process are used as initial value for a consensus run.

Join requests are treated as messages delivered with virtual synchrony guarantees. Upon reception of such request, the view change protocol is launched and a temporary view is decided among the surviving nodes. After that, each of them applies the join operations to construct the definitely installed view.

Lightweight groups The abstraction of lightweight groups (LWG) was introduced by Glade et al. [12] in the context of the Isis successor system, Horus, for tackling the scalability problem.

The underlying idea is the mapping of a large number of light groups onto a single heavyweight *core* group (HWG), so that LWGs share group state data with their maps. The HWG is used as communication transport for multicasts within its mapped LWGs. Since a membership change to the core group directly affects all mapped LWGs, the cost of reconfiguring them is much lower than in case they were maintained as entirely independent groups. A much larger number of groups can thus be supported.

The modular structure of Horus includes as its most basic components a multicast transport service and the *VSync kernel*, providing order guarantees to message delivery and virtual synchrony. On top of them, at the user–space level, resides the LWG subsystem, which dynamically manages the mapping between both levels, creating, changing and deleting core groups as necessary. A garbage collection process is needed to dispose HWG that are not serving any LWG. The mapping policy can be determined by the application or use the default setting in which a LWG is mapped to the smallest core group containing all of its members with a threshold for maximum extra participants.

A more recent design of a lightweight group service is described by Rodrigues et al. in [13]. Different to the original approach, a LWG does not need to specify its composition upon creation, but offers to the application the same interface of HWG and semantics of virtual synchrony. This design, also implemented in Horus, allows LWGs to be dynamically remapped to a different HWG as their necessities change. The service defines the policies for mapping and switching heavy groups. An external name service is used to keep track of the mapping.

To join a LWG, a process obtains the HWG mapping from the name service (if it did not exist, it is created) and sends a join message to all members of the HWG. The coordinator of the LWG triggers a flush protocol, in which all applications refrain from sending messages until the new view has been delivered. A similar procedure is followed when a process leaves the LWG.

Spread. Spread [14] is an open source group communication system providing Extended Virtual Synchrony [32] semantics over wide area networks. Spread uses a hierarchical structure in which user applications register as *clients* of long–living *daemons* running on each host. A group of daemons on a broadcast domain, typically interconnected by a LAN constitute a *site*, one of them acting as a gateway.

An overlay virtual network is constructed with links between all sites with active daemons. The membership service maintains information about the daemons and the link weights, in order to calculate shortest paths. Links between pairs of nodes are eventually reliable, and the failure model is crash/recovery.

Spread supports multiple groups, and uses the heavyweight group–lightweight group paradigm to maintain daemon and process membership, respectively, by means of a two layer structure executed at the daemon level [38]. The lower layer, daemon membership, copes with network partitions and merges, as well as with individual daemons failures and incorporations, and provides Extended Virtual Synchrony. The second layer manages lightweight group events, caused by joins or leaves of individual clients. Those are simply handled as notification messages, sent and delivered by daemons.

Moshe. Another approach to group membership on a WAN is that of Moshe [16], a membership service specifically designed for wide area networks. Moshe is characterised by a client–server architecture, in which servers are in charge of maintaining client membership, but do not participate of the group themselves.

Moshe relies on a *network event mechanism* specific for a WAN, which keeps servers informed about changes to the group and performs as an eventually perfect failure detector. This is realised by Congress [15], which provides approximate membership information. Congress assumes reliable FIFO channels and a strong complete failure detector. Its architecture is also client–server and it takes charge of diffusing membership changes among servers, without guarantees on the relative order of the notifications.

The specification of the membership problem solved by Moshe relaxes the requirements of the agreement to cases in which the networks stabilises and the notification service informs consistently. It does not prevent the delivery of certain views only to part of the servers, in periods of instability in which changes happen too fast. In particular, Moshe avoids the delivery of obsolete views, if it is known that the group has already changed.

When a server receives a network event notification, a fast agreement algorithm is launched, designed to terminate within a single round in benign cases. Each server multicasts a proposal to the others and when a consistent information has been received from all of them, each server notifies its clients about the installation of a view. In some cases the fast algorithm may block. Then a slow agreement algorithm is triggered by the block detector, and servers must exchange extra messages to synchronise their proposals.

Moshe notifies clients not only about view installation, but also about the beginning of a membership change. Congress and Moshe are integrated in the group communication system for wide area networks Xpand [39].

InterGroup. Another group communication system devised for scalability in wide area networks is provided by InterGroup [17], a partitionable and modular group communication system. The particularity of InterGroup consists in allowing each process to select the desired message delivery guarantees.

Reliability of underlying channels is not assumed, but InterGroup includes a module for reliable multicast.

The processes in the group are classified according to their activity into *active senders* and *receivers*. Only active nodes participate in the membership agreement, which is maintained with virtual synchrony guarantees by protocols based on those of Transis. Such protocol is launched when an active sender detects a failure of another member of its view or a non–active member starts sending messages. The receivers that are requiring total order run a lighter protocol. They receive information on the beginning and completion of the view change from some active sender to ensure virtual synchrony is respected.

A process may enter the receiver group by only contacting an active sender to learn about the current view. To become a member of the sender group, an active sender must also be contacted. This will broadcast a message to the group which, when delivered, completes the addition.

Weak Consistency Membership. An alternative approach to large–scale membership was presented by Golding and Taylor [40, 41]. The central idea of their specification was to relax the consistency requirements, allowing temporary inconsistencies, so to reduce latency and communication overheads. The specification guarantees only that after changes cease, processes will eventually converge to a single consistent view. Name services are cited as an example of applications that can tolerate the inconsistencies and make use of this service.

The system model assumes unreliable communication channels and the model of crash/no recovery for processes, although partitions may happen and remerge.

Each process maintains a view, formed by a list of processes, together with their status and a timestamp indicating when they entered it. The protocol is based on epidemic communication. In particular, it makes use of the specific time–stamped antientropy communication protocol, to ensure eventual consistency (all updates are eventually seen) and detectable consistency (a process can know when all the others have either observed a given update or failed).

Each process keeps vectors with the latest updates it has received from each other member, and with the most recent update each member has observed. Periodically a member of the view is selected to reliably exchange this information by means of an antientropy session. When a process wishes to join the group, it must first obtain sponsorship from a big enough number of members, in order to ensure failure resiliency. To leave voluntarily, a process sets its own status to *failed* and runs antientropy with some member. If a failure is detected, the status of the failed node is set to *failed* with infinity timestamp, so to reject outdated messages. This information is spread through the group by subsequent antientropy sessions.

2.2.3 Systems with Specific Needs

As technology develops, different models of distributed systems become available, for which distributed applications are developed. Distributed protocols have to fulfil specific requirements depending on the new environments and applications. To mention a few, we may name wireless networks and peer-to-peer systems, as examples of environments, and collaborative work as an example of new applications. Wireless networks have particular limitations, not shared by wired communications, which may hinder the guarantees of distributed protocols. The particular case of ad hoc networks is yet more demanding in that sense. Peer-to-peer systems, on the other hand, have goals related to eventual dissemination of information, and thus require relaxed guarantees from underlying services. Collaborative applications may show diverse requirements, between those of cluster–like systems and the relaxed semantics of eventual guarantees, and middleware for reliability support must cope with such a wide range.

Different approaches have been taken when developing membership solutions for these and other systems. Even new paradigms have been proposed, at least in theory, as *fuzzy membership* [42], as an approach to provide useful membership semantics for novel systems and applications. Here we review some of the membership services that have been developed for such systems in response to their particular features and necessities.

Collaborative Work

In collaborative environments, a number of computer systems participate of some form of cooperative work. Frameworks for collaborative computing typically provide an assortment of communication protocols. The applications that can benefit from these systems go from distributed databases or on–line discussions, to conferencing tools or video servers. Their requirements are also very different, including a variety of *Quality of Services*, and different levels of consistency for replicated data and distributed applications. Support systems should cope with such a variety of necessities. Their requirements from a membership service may thus differ from the most strict specifications.

Collaborative Group Membership. Pascoe et al. [43] propose a Collaborative Group Membership (CGM) approach, in the framework of the Collaborative Computing Frameworks (CCF) environment. The lowest layer of this framework provides a number of communication protocols. It also offers support for *sessions*, maintained as heavyweight groups, and *channels* which support different QoS. Failure detectin is done by the reliable multicast facility and a specific channel is devoted to reliable failure notifications and election protocols.

The basic idea of CGM is to take into account the possibility of partial failures in which a node does not fail completely but only in a subset of its systems. This is solved by running two levels of election. The first one, for membership removal, is launched when a failure is detected. The master then asks for votes and if a majority agrees on the failure, a new view is multicast. If not enough favourable votes are collected, a session election is performed, in which consensus is sought with respect to partial failures and a complex session image is built, with the failure report regarding every node and channel.

Caelum. Caelum [44] is a toolkit for the development of highly available groupware and collaborative applications, which includes a multimedia multicast service, a membership service and a module for session services, such as secure multicast or support for replication.

The membership service is hierarchical, based on Congress [15] and its incremental updates of membership composition. Therefore it does not guarantee that views are delivered in consistent order at every member. If an application requires such stronger consistency, it must implement the extra agreement on top of the service.

P2P Environments

Peer to peer systems have gained increasing importance as an alternative to client–server applications. They are constituted by a large number of loosely coupled nodes which form a totally decentralised entity. Data and control are completely symmetrical, and try to offer easy scalability and management, with an interest on self–organisation.

Typical tasks covered by P2P systems include content–based searching, service location, aggregation and efficient dissemination of information. Whatever the final applications, these systems include a basic layer in charge of group communication and membership, which provides the required guarantees to upper components implementing particular functionalities. Such requirements are however much less demanding than in cluster–like scenarios.

PeerCast. PeerCast [45] is a Peer-to-Peer system providing multicast dissemination of information with the model of End System Multicast (ESM). Each peer is equipped with PeerCast middleware, including layers for P2P network management and ESM management. PeerCast reliability is limited to continuation of the ESM service when some of the intermediate nodes fails. No guarantee is provided if the source of information crashes.

PeerCast does not employ a standard membership service with strict guarantees regarding view changes. A P2P membership protocol is employed to establish an overlay network, organised according to proximity of end-system nodes. The network is a distributed hash table of peers, with their identifiers and properties associated to a routing table and a list of neighbours, but peers do not have global knowledge about the composition of the network. At the level of ESM management, creation and subscription of multicast groups is handled. Continuity of the service in front of a failure is achieved by replicating the multicast service information across neighbours.

Newscast. Newscast [46, 47] is a peer-to-peer protocol for maintaining, disseminating and aggregating information in very large and dynamic networks. The system is supposed to consist of a very large number of nodes, each one with a series of attributes. Each node is running an agent which may generate news items that are to be disseminated across the whole network.

The protocol is epidemic and completely symmetric. Periodically nodes exchange information. Each peer knows only a fraction of other members, among which randomly selects one to exchange cached information. This includes membership information, as the addresses of agents, and application specific information. Joins are disseminated as correspondent addresses get exchanged together with other information posted by them. When a member fails, it stops sending news, and after a certain time, all references to it are removed from the system because of ageing. Thus, membership information provided to each peer is only partial, changes continuously and it is provided with no consistency guarantees. The *macroscopic* behaviour of the protocol is given by its statistical properties.

Astrolabe. Astrolabe [48] is a distributed information management service devised for large scale networks. Its goal is to monitor a set distributed resources that change dynamically, and to support the calculation of information aggregates that are required by higher level applications. It uses a relaxed consistency model, called *eventual consistency*, that guarantees only that if updates cease, aggregates will eventually be the same at the different agents.

For higher scalability, Astrolabe organises nodes in a hierarchy based on domains. The responsibility for the organisation corresponds to system administrators which assign names to zones. Its structure is a tree, whose leaves are nodes. Each host runs an Astrolabe agent. A gossip protocol causes pairs of agents to exchange state information, including membership. Failures are detected by excessive ageing of information from the representative of a certain zone. Since the network can partition or the tree can split due to wrong suspicions, each tree periodically multicasts probe messages to contact isolated trees.

Scamp. Scamp (Scalable Membership Protocol) [49, 50] is a probabilistic membership service designed for supporting gossip–based multicast algorithms. It does not provide the consistency guarantees of traditional membership services. Each node is provided with a partial view, which contains potential destinations of its gossipping and changes dynamically. Nodes also maintain a list of members from which they can receive gossips.

Scamp is a decentralised, peer-to-peer protocol. When a new node joins the group, by contacting a former member, a fixed number of nodes include it in their partial views. To do so, the contact forwards the information on the new member to nodes in its partial view, which depending on the size of their own views include the new member or forward the information again. To abandon the group voluntarily, a node must instruct the nodes that kept its identifier in their partial views to substitute it by nodes in its own partial view. After a given time without news of a certain node, it is removed from all partial views. This failure detection mechanism may conduct to isolation of nodes. To prevent this, periodic heartbeats are sent by every member.

Nor global view, neither consistency guarantees are provided by Scamp. Its goal is to support reliable multicast by gossip algorithms, in which each node gossips to its partial view. Its robustness features arise purely from its statistical properties.

Wireless, Ad Hoc Networks

Wireless communication technology has boosted the development of mobile computing. The support of distributed applications for these environments requires taking into account different aspects, including possible message losses due to unreliability of wireless links, change of location of mobile nodes, and disconnected operation. In typical systems, a certain fraction of mobile nodes connect to some wired or stationary hosts, that can be used to deploy centralised solutions or at least to make use of the more benign features of traditional networks.

Ad hoc networks constitute a particular case of wireless systems, in which no fixed node exists and the topology of the networks is unknown. Few approaches exist to developing specific membership services for ad hoc networks.

Localised GMS. Briesemeister proposes a Localised Group Membership Service [51], for maintaining membership information in a system composed of an unbound number of wireless nodes that can move freely. The system is modelled as asynchronous, the model of crash/no recovery is assumed for failures and the network may be subject to partitions and message losses, but not to delays. Each process is given an identifier and is characterised by a space–time location, which may change during the execution.

The goal of LGMS is not to provide members with consistent global views, but maintaining at each host information about its neighbourhood membership. To do so, LGMS is deployed on top of a Neighbourhood Service, in charge of keeping an up-to-date list of adjacent hosts, by means of periodic heartbeats emitted by each process. A process becomes a member of a group when the application on top of LGMS invokes its *join* method. Views installed by LGMS at each host are formed only by members and neighbours of that particular host.

Safe Distance. Huang et al. [52] have proposed a strong partitionable membership service for ad hoc networks. The service provides each host with consistent membership information and message delivery guarantees in front of mobility induced changes of connectivity. The assumed system model is partially synchronous, with reliable network channels and bounded delays. No messages are ever lost due to network congestion. Also, crash failures are not tolerated, and the only communication failures are caused by nodes moving out of each other's range.

The service specification includes a strong requirement regarding membership delivery, namely *Sending View Delivery*, which must hold even in the occurrence of partitions. To ensure such guarantee, the protocol relies on the concept of *safe distance*, which is the maximum distance between a pair of nodes such that any communication task can certainly terminate, assuming the nodes move randomly with maximum speed v. This notion is used to predict disconnections in a timely manner and consequently decide whether a host may or not be included in the group in order to ensure the desired guarantees. Moreover, every node is supposed to have a localisation service, providing it with information about its physical position.

The protocol is centralised and makes use of an underlying discovery protocol. There is a leader, deterministically elected per group. Each node periodically reports its position to the leader, and also any discoveries of potential merge candidates. The leader decides on configuration changes, leading a merge protocol if neighbour safe groups are discovered or issuing a splitting order if movement of group members causes the configuration to stop being safe. To exclude part of the group, a message is broadcast to all members, containing the new membership information. Then each of them stops sending messages, broadcasts a flush message and waits for identical messages from its fellow members, at which moment the new configuration is installed and message sending is resumed. Merging is performed in a similar way, but it is preceded by a negotiation between both group leaders to reach an agreement about the participants and the coordinator.

2.3 The Formal Specification of the Problem

As discussed above, none of the proposed specifications for the group membership problem is universally adopted. The variety of proposed algorithms responds to the diverse requirements of their potential application systems, as there is no universal semantics that helps all of them. Moreover, most of the originally defined membership services were designed for different group communication systems, and their specification was joint with that of group communication guarantees.

An important unification effort was done by Chockler, Keidar and Vitenberg in [22], a comprehensive survey on Group Communication Specifications. In that work, a number of existing systems are reviewed and their properties are enumerated in a unified formalism. Membership properties are separated from those of multicast, and a set of basic properties is identified that are common to every analysed membership service. Differences between primary component and partitionable specifications are also enhanced. The fulfilling or not of additional properties confers each system its particular characteristics. Some recent works, as [53], take the set of basic properties of [22] as the basis for membership specification.

An alternative specification of a partitionable group membership problem, completely independent of multicast guarantees, is given by Babaoglu et al. in [54, 10]. In this one, special emphasis is placed in the differences that the partitionable character imposes to both the specification of the membership problem and the system model, including the properties of the failure detector.

A more recent work by Schiper and Toueg [55] proposes a different approach to the specification of the group membership problem. In there, the primary component group membership problem is specified as a particularisation of a more fundamental problem, the *set membership problem*. These problems regard the agreement among a group of processes about the members of a set, which in the case of group membership is a subset of the process group itself, but they completely decouple the specification from the determination of correct processes, as that is another problem, to be solved by the failure detector. Their specification, therefore, does not include terms as *correct* or *failed*, but relates changes of view to orders issued by member processes, regardless of their reason. To our knowledge, no similar attempt exists to specify the partitionable membership problem.

It is important to notice that all formalisations above correspond to the *strictly* consistent group membership. For more relaxed membership semantics as the ones described in the previous section, neither a unified formalisation nor a compilation of properties as the one in [22] exists. On the contrary, most services are usually specified in a non–rigorous way, or at most use specific notations.

Chapter 3

The Group Membership Architecture

In the rest of this work, we will specify different membership services and their implementations for various scenarios. The cornerstone and reference for every kind of membership semantics discussed here will be the strongly consistent semantics provided by traditional membership services. This will be referred as *basic membership service*. As discussed in the previous chapter, the specification of the group membership problem is not unique. Nevertheless, for the traditional membership services, a set of fundamental properties were identified in [22]. In this chapter we discuss these properties in the context of the system model they apply to, and we compile from them the basic specification to be used as reference.

Although the multiple existing specifications and realisations of membership services provide their information with particular formats, we postulate that any existing membership service, including those that correspond to extended or relaxed semantics, can adapt to a common generic interface. This chapter describes in detail such interface, and also the modular membership architecture that characterises this work.

3.1 The Basic Membership Problem

3.1.1 System Model

We will be assuming a partitionable system formed by N processes that communicate by message passing. Since the system is asynchronous, there is no known upper bound on message transmission time. Neither there is a global system clock, nor synchronised local clocks at the various processors.

The process failure model is crash/recovery, i.e. a process fails by simply stop executing actions, but this failure is not detectable for the others. After a crash the process may recover and then rejoin the group. Every process is equipped with a local unreliable failure detector module, which behaves as eventually perfect.

Network channels may cause message loss, disorder and duplication, but they respect the integrity of messages, and they provide the semantics of *eventually reliable channels*. If a process p sends a message to a process q and both the sender and the receiver remain correct, then the latter eventually receives the message.

The consideration of crash/recovery model has certain implications on some elements of the model, as discussed in [25] in the context of solving consensus in this model. Regarding failure detection, the strong completeness property FD.1 would require the safe distinction between process that fail and recover, no longer crashing, and those that crash infinitely often and thus cannot contribute to the algorithm. To overcome such difficulty an alternative completeness property is enunciated.

Property FD.6 (Recurrent Strong Completeness.)**.** *Every* bad *process*[1] *is infinitely often suspected by every correct process.*

Failure detectors satisfying this property and eventually weak accuracy define the class $\diamond\mathcal{S}_r$, which allows consensus to be solved in the crash/recovery model. In practice they can be implemented by periodic heartbeat plus *alive* messages sent by recovered processes. A class $\diamond\mathcal{P}_r$ can be analogously defined, by combining eventually strong accuracy with property FD.6. We will thus consider the processors in our system to be equipped with a failure detector behaving as $\diamond\mathcal{S}_r$ or $\diamond\mathcal{P}_r$.

In order to recover its state after a crash, processes should have some stable storage. It is possible to solve consensus without that facility, provided that a minimum number of processes never crash (more than the number of failed processes). Since we consider every process capable of failing, we also assume that each of them is equipped with stable storage.

The network may partition due to link failures. Virtual partitions may also occur due to wrong failure suspicions. We want the algorithms to make progress in each partition, regardless of their majority character. Also, the persistence of a majority partition along the whole execution cannot be guaranteed. On the contrary, the network may split into small disjoint groups. After failures are repaired, disjoint partitions can remerge and the majority character be recovered.

[1] *Bad* processes are those that either fail and never recover or crash infinitely often.

3.1.2 Basic Specification of the Group Membership Problem

For the rest of this work, we adopt as the *basic specification of the partitionable group membership problem* that results from the basic set of membership properties in [22]. This is the approach adopted also by more recent works, as in [53]. For a specification that is completely independent of group communication guarantees, an agreement requirement must be added. For this property, we adopt the enunciate given in [10].

Regarding more relaxed semantics, appropriate specifications will be introduced for other membership problems later in this work, by relating them as much as possible to the basic specification.

The specification of the basic partitionable GMP is contained in a set of four safety properties and one liveness property, as follows.

Safety Properties

Property GM.1 (Self Inclusion). *A process is member of each view it installs.*

Property GM.2 (Initial View Event). *Each event occurs in some view context.*

Property GM.3 (Local Monotonicity). *Two committed views are never installed in different order by two different nodes.*

Property GM.4 (View Agreement). *(i) A process does not install a view unless the immediately previous one has been installed by all other processes contained in both views.*

(ii) If some node contained in a committed view does not install it, or installs a different view as its immediate successor, all nodes in the former view will eventually install a new view as the immediate successor to the first one.

The property of *View Agreement* requires that there is agreement among members of a partition about the composition of the view. In particular it states that, if it is not possible for all correct members of a view to install it, or if some of them sees another change, all the remaining ones will also see a change. The basic specification in [22], not being completely decoupled from the multicast specification, does not contain this property, as it can be derived from the basic multicast properties. Since we need a stand-alone specification of the group membership problem, we include the property GM.4 explicitly, as enunciated in [10].

Liveness Properties

Property GM.5 (Membership Precision). *If there are stable components in the system, i.e. subset of processes that remain correct and connected, the same correct view is installed as the last one in every process of the same stable component.*

This property tries to guarantee that the system reflects the real system state, as much as possible, and to preclude trivial solutions in which the membership views split arbitrarily. In the considered model, the liveness properties of the membership information depend on the eventually perfect execution of the failure detector, and on some assumptions on the system dynamics. The term *stable component* denotes a subset of processes that remain correct and mutually connected, whereas they are unreachable from any other nodes that are not in the same subset. Thus the liveness property GM.5 is conditioned to the fact that such components survive long enough for the failure detection picture to stabilise and the membership protocol to complete the view installation.

3.2 Generic Interface

We express the interface between the membership service and other components with a client–server structure, in which any component making use of membership information acts as a client of the membership service. This architecture allows multiple system components or user applications to make use of membership information, and corresponds with some of the most recently developed membership services [16, 14]. We postulate that every existing membership service can be adapted to the same generic interface, contained in figs. 3.1, 3.2.

```
public interface IMembershipMonitor {
  void joinGroup ();
  void leaveGroup ();
  void registerListener (int type, IMembershipListener listener);
  void unregisterListener (int type, IMembershipListener listener);
}
```

Figure 3.1: Generic interface of an arbitrary Membership Service.

The *basic* semantics of the interface of `IMembershipMonitor` is the following.

- `joinGroup`; the first time this method is invoked, the membership monitor running in the local host should try to contact other members in order to be accepted by the group, if it exists, or to promote the formation of a new one;

```
public interface IMembershipListener {
  void notify(MembershipEvent evt);
}
```

Figure 3.2: Generic interface of a Membership Listener.

- `leaveGroup` forces the node to leave the group; it is invoked by a node to voluntarily leave the cluster, for instance when it is about to switch off;

- `registerListener` lets the membership service know about a new component interested on a particular type of membership event so that from this invocation on the component will be correspondingly notified each time a `MembershipEvent` of the specified type occurs. This method must be invoked once per component and type of event;

- `unregisterListener` announces that a previously registered component is no longer interested on a particular type of `MembershipEvent` and thus it causes the monitor to stop notifying that type of event to the particular client.

Such structure allows various components to be served by the membership service, thus receiving consistent membership information.

Client applications, on the other hand, should be registered as `IMembershipListener`, implementing a callback that will be used to notify them about membership information, and specifying at registration time the type of event they are interested in. Membership views are delivered to registered applications as `MembershipEvents` by means of the callback `notify`.

The interface is completely general, as it allows the membership service to deliver different types of membership information, by defining types of `MembershipEvents`. Although the basic specification of the group membership service regards only the installation of views, and thus would fit a single type of `Event`, there exist services and specifications that deliver additional information. E.g. Moshe [16] notifies clients not only about installed views, but also about the beginning of membership changes, which may be not followed by the corresponding view installation if successive changes occur. The Optimistic Virtual Synchrony model [33] employs the installation of *optimistic views* with a tentative set of members once the need for a view change is detected. This is used by applications to avoid blocking and do some progress until the definite view is installed. These features nicely fit in the above interface by defining extra types of `MembershipEvent` for optimistic or tentative views.

3.3 Modular Architecture

A common characteristic of all the services described in this work is their modular architecture. This is not a new approach. On the contrary, recent works in the field of group communication systems [56] point out the convenience of a modular architecture, where each service is devised as an autonomous, self-contained entity. In [56] the system includes independent services for group membership, failure detection and various communication guarantees. Moreover, each service offers standard interfaces to the others, and to the application. Such a scheme allows the easy integration or replacement of individual components and contributes to the usability of the system by the application developer.

Some of the most recent implementations of group communication services already adopt a modular structure. It is the case of Jgroups [57], Appia [58] and Cactus [59]. In those services, nevertheless, the different components are not designed to be used independently.

Our aim is to provide generic membership services that are usable by any component or application. Moreover, the designed membership services are decoupled from other services, including group communication and failure detection. In our scheme, communication among these interacting parts happens only through general interfaces, by means of `Event` notifications.

Specialised `Events` are produced by the different components and notified to their registered listeners. As an example, a failure detection component should produce `FailureEvents`. The membership service should be registered as a proper `Listener` to receive and use this information.

Fig. 3.3 shows the proposed architecture for our basic membership service. Contrary to other approaches, the membership service does not hide the information from failure detection from other components. As noticed in [60], some applications may progress with only information about unreliable failure suspicions, and may thus use direct input from the failure detector component. Moreover, this architecture gives us more flexibility to employ a different mechanism for failure detection depending on the system characteristics and conditions, without varying the implementation of the membership service itself.

The bottommost layer in the proposed architecture corresponds to the `UnreliableTransport`. since we do not make reliability assumptions on the transport, our membership service makes direct use of this layer, which abstracts the real network from any other component in our system, providing the very simple interface in 3.4.

To illustrate the usage of this structure, we also show in fig. 3.5 a possible architecture for a system providing various services to the potential applications.

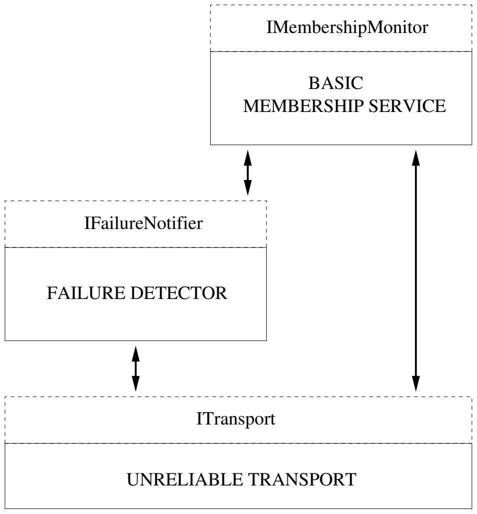


Figure 3.3: Architecture for the basic membership service.

```
public interface ITransport {
  void send (IDestination dest, Message msg);
  void registerMessageHandler (int type, IMessageHandler handlr);
  void unregisterMessageHandler (int type, IMessageHandler handlr);
}
```

Figure 3.4: Interface of the bottommost transport.

The picture shows explicitly the interfaces offered by every component to applications and higher layers. Communication with components below also occurs through the generic interfaces, which are not shown in the figure.

In this example system, a reliable transport protocol, a group communication system and a certain user application, B, will be using membership services. All of them then implement the `IMembershipListener` interface. On the other hand, the Membership Service, and user application A act as listeners of the Failure Detector, whereas the Group Communication System, the Reliable Transport, the Membership Service and the Failure Detector itself, together with user application D, make direct use of the Unreliable Transport, and must then implement the interface of `IMessageHandler`.

The generic interfaces and the modular architecture of the membership and other services are part of the more general structure of HAMS system. HAMS, which stands for Highly Available Middleware Systems, is a middleware architecture devised to provide high availability support to legacy systems, by means of a configurable set of services and protocols which can be chosen upon configuration so to implement the necessary guarantees. This support is now being used in several projects [61, 19, 18].

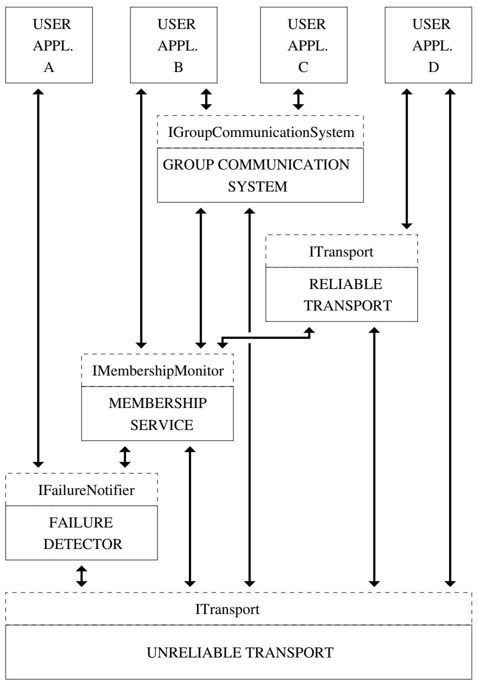


Figure 3.5: Possible architecture for different services and applications.

A detailed description of the whole architeture escapes the scope of this work and can be found in [20].

Chapter 4

The HMS Protocol

In this chapter we present the design of HMS (HAMS Membership Service) [62, 63], a partitionable group membership protocol that implements the basic specification of the problem 3.1.2. As an extra feature, HMS provides also the property of majority view uniform agreement, that will be discussed in detail later. HMS constitutes the basic building block for some of the other membership services discussed in this thesis. The following sections discuss in detail the main aspects of the HMS protocol, including its state, specific messages and operation.

4.1 The System Model

The HMS protocol is designed to maintain the membership of a cluster with the strongest consistency guarantees and full partitionability. The protocol aims to monitor a group of nodes of well–known identities, thus in the rest of this chapter the term *node* will substitute that of *process* when referring to group members.

As the rest of protocols in this work, it assumes an asynchronous system, formed by a set of nodes that communicate over unreliable network channels. The nodes are assumed to be part of a well–defined, preconfigured set, of a maximum size of several tens of nodes. Each node has a unique identifier, which is assigned in advance and known a priori by every other member of the system.

Cluster nodes are typically connected over a LAN, and they remain correct and connected most of the time. Nevertheless, they are allowed to fail by crashing, and to recover afterwards. The network is also unreliable, so that messages may be lost, disordered or arbitrarily delayed. The protocol thus handles these situations however frequent they are.

Each node is equipped with a local failure detection module, providing unreliable suspicions information to the membership protocol. We assume that the failure detector behaves as $\diamond\mathcal{P}_r$. Despite the theoretical impossibility of the liveness properties in the purely asynchronous model, in practical systems such a component can be implemented [10] provided the system exhibits a reasonable behaviour. Therefore our system model includes an assumption on the dynamics, namely that there exist failure-free intervals of time long enough for the failure detector to stabilise and a full reconfiguration to take place. This is, many nodes may happen to fail during a certain period of time, and these failures can be completely arbitrary, but such a situation cannot last forever. At a certain moment the *correct* nodes will stay alive for a long enough interval of time, so that the membership monitor can reach a stable configuration. In the considered scenario this is a natural assumption, enough to make a live service implementable.

4.2 Basic Elements

This section describes some of the main elements for the HMS protocol. These include the view identifiers and the state maintained at each member node. Since the failure model is crash/recovery, the usage of stable storage is also required.

HMS is an asymmetric protocol, in which a master leads the proposal and installation of every view, as well as the procedures for partition merging.

From another point of view, being partitionable, HMS includes two modes of operation, for majority and minority, respectively. The particular operation mode decides the view identifier, and the part of the state that is to be used.

4.2.1 View Identifiers

Each installed membership view is given a unique identifier. They are used to label messages that depend on view context and also play a crucial role for ensuring the property of *local monotonicity* GM.3.

The view identifiers of HMS are chosen from the set

$$\mathcal{V} \equiv \{(a, b, c) \mid a \in \mathbb{N} \text{ and } b,\, c \in \mathbb{N} \cup \{-1\}\}\,.$$

The three integers a, b, c that form the identifier are defined by the following rules.

- The first component, a, is a non-negative integer which plays the role of the majority view identifier. When the group is a majority, i.e. it contains more than one half of the preconfigured set, the view identifier is of the form $(a, -1, -1)$. Thus during majority operation a is the only significant part of the view identifier. The Master increases its value each time a new majority view is proposed.

- The second component in a view identifier of the form (a, b, c), identifies the Master of a minority partition originated from version a of the majority group. When the group loses the majority condition, a can be no longer changed. The minority Master will construct new view identifiers maintaining the last installed value of a, setting b to its own identifier.

- The third component, c, serves as the identifier of minority views with common majority root and minority Master. It is set to 0 every time b changes, and then it increases monotonically with every new proposal of the same minority Master.

- The criteria explained above do not suffice to generate unique view identifiers. Within the minority group, repeated partitions may arise and the minority master can be repeatedly substituted. If the node acting as minority master fails and is substituted, and then it recovers, each time it happens to recover mastership, it would generate an identical view identifier. To avoid this problem, the second component, b, of the view identifier keeps track not only of the identity of minority master, but also of the *incarnation number* of its mastership, k, i.e. the number of times the current minority master has assumed mastership since the partition occurred. Thus, when a certain node of identifier m substitutes the minority master for the k-th time with the same majority root a, it constructs a new view identifier by setting $c = 0$ and $b = m + k \times N$, being N the preconfigured number of nodes in the system.

The partial order relation within the set of view identifiers allowing the implementation of property GM.3 is defined as follows.

Definition (Partial Order of View Identifiers)**.** *Let V_1 and V_2 be two different views, with respective identifiers (a_1, b_1, c_1) and (a_2, b_2, c_2), such that $b_1 = m_1 + k_1 \times N$ and $b_2 = m_2 + k_2 \times N$.*[1] *We will say that V_1's identifier is lower than V_2's, and write $V_1 < V_2$ if any of the following conditions is fulfilled.*

1. *$a_1 < a_2$ and $b_1 = -1$. In case $b_2 = -1$, this is the straightforward comparison of two majority identifiers. If $b_2 \neq -1$, then V_2 is a minority view originating from a majority view that followed V_1.*

2. *$a_1 < a_2$, $b_1, b_2 > 0$ and $m_1 = m_2$. In this case, V_1 and V_2 are minority views proposed by the same master in completely different partitions, so that the majority was recovered in between.*

3. *$a_1 = a_2$, $b_1 = c_1 = -1$, while $b_2 > 0$. In this case, V_2 reflects a minority partition originating from V_1.*

[1]Notice that m and k can be easily calculated from b as $m = b \mod N$ and $k = \frac{b-m}{N}$.

4. $a_1 = a_2$, $b_1 = b_2$ and $c_1 < c_2$. This case corresponds to two views of the same minority partition, and proposed by the same master in the same mastership incarnation.

5. $a_1 = a_2$, $m_1 = m_2$ and $k_1 < k_2$. This case corresponds to two views of the same minority partition, proposed by the same master in different incarnations.

Any pair of view identifiers satisfying none of these conditions can be considered concurrent, as direct comparison is not possible between them. Sometimes, there may be causal relations between views with *concurrent* identifiers. This may happen when the information about the history of installed views kept by a node and the whole view composition allow the deduction of such relations, which are not contained in the view identifiers.

If we were only concerned with majority views, the first condition would suffice to implement the monotonicity property. Since the protocol must also handle minority partitions, however, the monotonicity must be guaranteed by the algorithm itself. The partial order relationship between view identifiers allows also the partial comparison of minority views, and results useful to discard obsolete messages.

4.2.2 State

Although the specification of the full state maintained by the HMS protocol at each node rather corresponds to the formal description of the algorithm, there are a number of fundamental data structures that will be extensively used in the informal description of the protocol operation. This section details the most relevant elements of the state required by HMS.

Memory

The most of the state is maintained in main memory, hence it is volatile. In particular, each node maintains two lists of views.

- The *strong list* is a list of majority views in the different stages of confirmation. Each list entry consists of a view and its confirmation status. The status can be `prepared` if the view has just been proposed or received from the master, and its confirmation is pending. If agreement has already been reached, so that the view has been installed, its status is `committed`. After each node has confirmed the commitment, the view can be promoted to the `released` status.

  The list keeps views sorted according to the total order of their identifiers. New views are added to the end of the list, satisfying the following conditions. There cannot be more than one view in the `prepared` or the `committed` status in the list. If there is a `prepared` view, then it is the last one of the list, and `released` views cannot be preceded by views in any different status. A view cannot get the `committed` status unless it was the last `prepared` one.

Every `committed` view eventually reaches the `released` status. At this point we may choose to leave the view in the list with a definitive `released` flag, or to remove it from the list to save resources. This is irrelevant from the point of view of the algorithm, so we will just refer to this stage of confirmation saying the view is `released`.

- The *weak list* is a different, alternative list used in a completely analogous way to the previous one when the node is in a minority partition, since the *strong list* is only used during majority operation. In the *weak list*, the same conditions are fulfilled, regarding the placement of views, with the only difference that in this case the sorting does not fulfil a total order relation of view identifiers, as this does not exist for minority views. Instead, views are sorted according to their causal relations.

Stable Storage: The Majority History

Since the failure model that is to be supported by HMS is that of crash/recovery, and given that HMS is meant to tolerate up to N failures of nodes, it becomes necessary to employ some stable storage that enables part of the state to be reread after a crash and a subsequent recovery. In the HMS protocol this is achieved by the *majority history*, a consecutive list of committed majority views, kept in stable storage to allow recovery of majority after partitions merge. Every majority view is saved to the *majority history* when it becomes `committed` in the *strong list*.

When a node starts, it checks whether its local *majority history* is non–empty, as will happen when the node is recovering after a crash or a stop. In such case the node can recover the latest information about the majority before starting the protocol.

If the protocol works for an indefinitely long time, the size of the *history* may become extremely large. To keep the size of this list finite, there must be a subprotocol taking charge of eventually cutting away the oldest, needless part of the list. This can be safely done after all preconfigured nodes get simultaneously connected and confirm a common majority view.

Besides the majority history, the stable storage is also used to keep track of how many times a given node has assumed mastership after a partition from the last installed majority view.

4.2.3 Messages

The protocol defines a set of dedicated messages that are exchanged by nodes in the different stages of operation of HMS. Each protocol message is labelled with the identifier of its originating view, and carries the node identifier of its sender. The precedence order relation defined in the set of view identifiers is used for recognising and discarding obsolete messages. Besides, different types of messages carry specific arguments as detailed below.

- **Setmem(**V**, . . .)** is sent by the master to propose the installation of a view V, and must be received by all the nodes in the group. It contains the proposed view together with some information on previously proposed views.

- **Step(**n_{step}**)** is sent by the master of the group to all members, in order to conduct the protocol of view confirmation. It must contain the number of the current step, $n_{\text{step}} \geq 1$.

- **Ends(**n_{step}**)** is sent by the members to the master, as acknowledgement and reply to **Setmem** and **Step** messages. It contains as argument the number of the **Step** message being answered, or $n_{\text{step}} = 0$ if replying to a **Setmem** message.

- **ChangeM(**V_{last}**)** is sent to all nodes in the group by the master's substitute, once it has detected a failure of the previous master, in order to collect information on the last view installed or proposed by the old master. It should include some information on the status and composition of the last view the substitute received from the old Master, V_{last}.

- **View(**V_{last}**, . . .)** is sent by each of the members of the group to the self-proposed master's substitute, as a reply to the previous message. Each **View** message contains information on the last view or views received from the old master by the sending node.

- **Join(**V**,** V_{M}**)** is periodically broadcast by the master of a minority group, in order to locate and join possibly disjoint partitions. The mesage contains the composition of the group which proposes the joining, V, together with the last majority view it has known about, V_{M}.

- **Joined(**Vid_{V}**,** $Vid_{\text{V}_{\text{min}}}$**, . . .)** is sent by a majority master when it receives a **Join** request from a minority group, V_{min}. It is addressed to all the nodes in the minority partition, in order to get them joining the larger group, and contains the identifier of the majority group, Vid_{V}. It also carries the minority view identifier and, if required, information about majority views that the smaller group is missing.

- **Ready**(Vid_{V}) is sent by nodes in a minority partition to a majority master, in reply to the previous message. It includes the view identifier received with a Joined message.

- **Update**($views$) is sent to minority groups in order to bring their majority history up to date. It carries as argument a list of views, $views$, containing all the information of the majority history missed by the minority group.

4.2.4 Membership Events

Membership services are devoted to notify changes to other components. Therefore they must produce some kind of output to the user components or applications. This is what we will call *membership events*. In the architecture described in the previous chapter, they correspond to the information communicated to every `IMembershipListener` by means of `MembershipEvents`.

The fundamental event in a classical membership service is obviously that of view installation. All classical services produce such output. For some of them it is the only produced output. Others, on the contrary, produce also notifications about *tentative* views.

The HMS service produces these, but also other kinds of membership information. We distinguish the following *membership events*. According to the generic architecture in which our implementation is to be deployed, we have assigned each of them a particular type of `MembershipEvent`.

- `prepare`. Occurs when a view is added to one of the lists with state `prepared`. In the scheme of our generic interfaces, it should be notified by means of a specific type of `MembershipEvent`, namely MBSHIP_CHANGE, having as argument the just prepared view. It can be interpreted by external components as a tentative or in–progress view installation.

- `commit`. Occurs when a view status is set to `committed` in the corresponding list. It corresponds to the classical notion of view installation at the local process. The notification to higher level components takes place by means of a MBSHIP_VIEW event, necessarily containing as argument the composition of the view.

- `release`. Occurs when a view is labelled `released` in the corresponding list. It can be notified to higher level applications by means of an event of type MBSHIP_RUNNING, which should be interpreted by the receiving component as a signal that the last committed view has been confirmed by all its participants.

- `upcommit`. This is a special type of event, related with the management of the majority history by the HMS protocol. This guarantees the uniformity of confirmed majority views across the whole system, which imposes the necessity to update one node's information about the majority history of the whole system. During this process, old majority views that were confirmed by the group are saved to the local stable storage, and may be communicated to higher level `IMembershipListeners` by means of a specific type of `MembershipEvent`, MBSHIP_UPCOMMIT.

4.2.5 Underlying Services

The operation of the HMS relies on the existence of a few underlying services which provide a certain input to the protocol. Some of them have already been mentioned when describing the system model. Here we focus on their interaction with the protocol, i.e. the information they provide to HMS and the interface they offer.

- Unreliable transport. This component offers the basic transport interface of fig. 3.4, with a `send` method that allows any registered component to address messages to particular destinations. It also allows the HMS component to register as handler of its specific messages, so that when one such message is received by this transport, it will be handed to the HMS component by means of a `receive` event. Notice that no reliability is ensured by the underlying transport, so that any protocol message can be lost. In order to avoid the algorithm blocking, reliable point–to–point delivery can be explicitly implemented for each message whose loss can prevent the protocol completion. In order to ease the description and realisation of the protocol, however, more general primitives `rsend` and `rmcast` are implemented as support for our protocols. The `rsend` primitive guarantees reliable point–to–point delivery by periodically resending messages until they are acknowledged by the destination or the latter is excluded from the membership group. The `rmcast` primitive does the same for multiple destinations. These primitives are used in the flowing description of protocols together with the basic `send`, `multicast` and `broadcast`, that offer no reliability guarantee.

- Unreliable failure detector. This component, the local module of a distributed $\diamond\mathcal{P}_r$ failure detector, will notify the HMS protocol about suspected nodes in the system.

- *Exclusion mechanism.* Since the component above is not reliable and can make mistakes, it is possible that the HMS protocol at some node excludes a certain member based on the failure detector fake suspicions. Nevertheless, the guarantees of the failure detector do not include symmetry, i.e. the excluded member is not guaranteed to detect its excluder as failed. In some situations, this might lead wrongly excluded nodes to stay indefinitely in an obsolete view from which they had already been removed by the others, thus blocking the HMS progress.

To avoid this situation a *symmetrisation* mechanism must be added, that notifies a node when it has been excluded from a view. This can be implemented by various mechanisms, which can be very simple (based on periodic beaconing) or more complicated (to reduce the risk of false exclusions due to message losses). Such component could be part of the HMS protocol or even part of the failure detector, enriching it with a symmetry property, but for simplicity, we opt for a decoupled component, that we will call *exclusion mechanism*, since we are not interested in the implementation details. In any case, we will consider that our failure detector is complemented by this type of module, so that our membership protocol receives failure notifications from the basic unreliable failure suspecter but also from this component, which runs only as an auxiliary service of HMS. The only guarantee required from this service is the following.

Property EM. (Exclusion Mechanism)**.** *If a process p excludes another process q from its view and q is operating, then eventually q will receive a notification about the failure of p.*

Notice that this property is easy to achieve in most systems. It would be enough to piggyback the failure detector messages with the current view. Then, from the received information, a node can exclude itself if it detects that a newer view not including it has been installed by (part of) its former group.

4.3 The Protocol

Our most formal specification of the HMS protocol was made in the formalism of I/O Automata [64, 65]. Such specification is presented in and appendix B.1 (a previous version can also be found in [66]). Nevertheless, for a better understanding of the protocol, we include in this section the specification in terms of states and transitions.

Figure 4.1 illustrates the states in which the protocol may be divided, and the transitions among them, driven by the messages enumerated above, as well as by events from the failure detector. In this section we describe the role of each state while the next two sections detail in an informal way the algorithms run in each state to ensure the guarantees of HMS. A more formal description is shown in the figures 4.2-4.11. For these algorithms, local variables are to be understood as `static` variables in C, i.e. their value is saved in between invocations. Their initialisation is carried by the **init** block in each algorithm.

4.3.1 States

- SINGLE: In this state a node is the master and only member of a singleton view. It is the initial state, as the node is alone at startup.

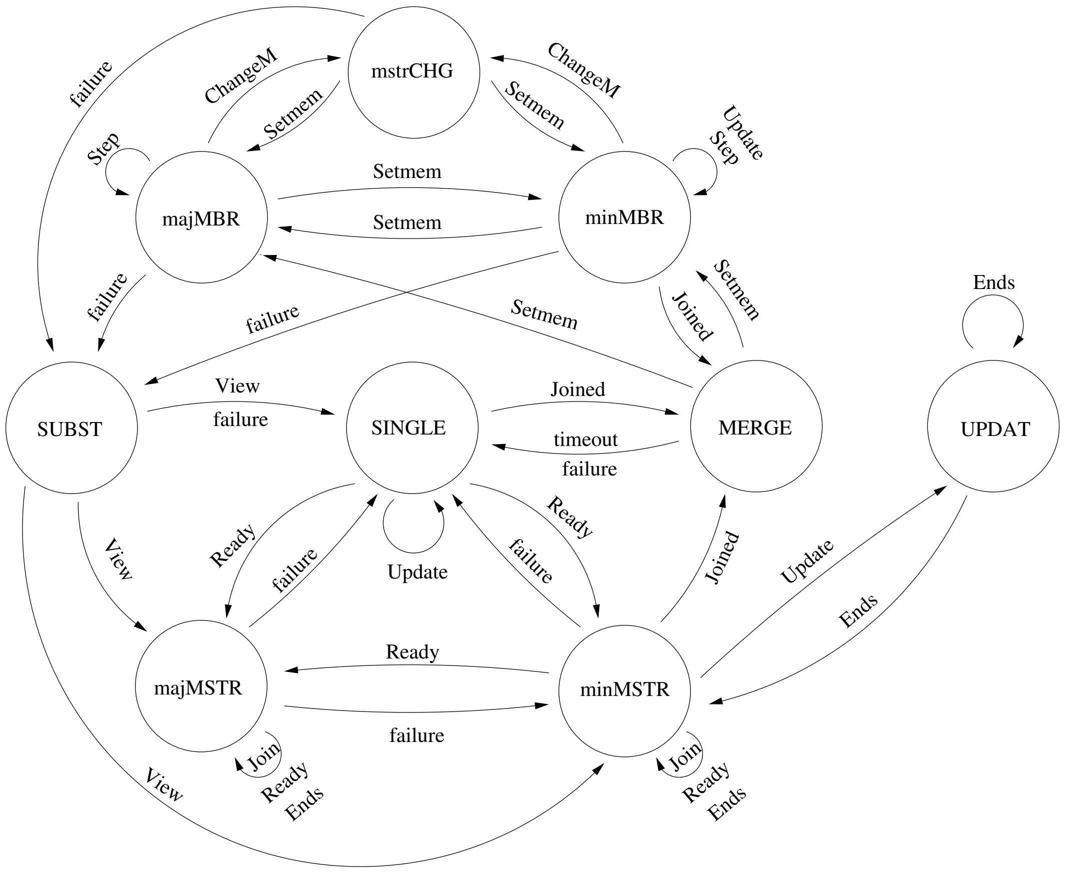


Figure 4.1: States and transitions of the HMS protocol.

- majMSTR: In this state the node is the master of a majority group, in charge of leading the installation of new views, as other nodes fail, and the merging of minority partitions if they try to join.

- majMBR: It is the state of all non–master members of any majority group during regular operation, i.e. not running the protocol for master substitution.

- SUBST: In this state, a node that detects the failure of the former master and decides that it has to substitute it, according to deterministic rules, tries to lead the master changing phase among the survivors.

- mstrCHG: Those nodes that are not to take the master's place during the change stay in the mstrCHG state while the substitute collects the necessary information to compose a new view.

- minMSTR: When the group is formed by less than one half of the preconfigured nodes, the master stays in the minMSTR state to lead minority view changes and try to probe for disjoint partitions to merge with.

- minMBR: Non–master members of a minority partition stay in the minMBR state until the group starts merging with a larger partition or a minority master change takes place.

- MERGE: It is the state adopted by all members of a minority partition after they have contacted the master of a larger group and are attempting to join it.

- UPDAT: A minority master enters this state when, as a consequence of its probing for larger groups, it receives a piece of the majority history that this partition is missing. In this state, the master suspends any attempt to merge other groups, neither it accepts such merging from other masters, until all its subordinate members have applied the set of updates to their majority history.

```
algorithm HMS
type
  state_t = {SINGLE, minMSTR, minMBR,
     majMSTR, majMBR, SUBST,
     mstrCHG, UPDAT, MERGE }
  view_t = (view_id, master, mbrs)
  view_st_t = { PREPARED, COMMITTED,
     RELEASED }
var
  state : state_t
  strong : list of (view_t,view_st_t)
  weak : list of (view_t,view_st_t)
  current : view_t
  lastPrep : view_t
  majHist : list of view_t
  next : view_t
  substit : node
  failed : list of nodes
begin
  strong := empty;
  weak := empty;
  state := SINGLE;
  next:=null;
  substit:= null;
  failed:= empty;
  majHist, lastPrep initialize
     from file;
  while true do
     case state of
     SINGLE:    single;
     minMSTR:   minmstr;
     minMBR:    minmbr;
     majMSTR:   majmstr;
     majMBR:    majmbr;
     SUBST:     subst;
     mstrCHG:   mstrchg;
     UPDAT:     updat;
     MERGE:     merge;
     esac
end;
```

Figure 4.2: Basic algorithm of HMS.

```
algorithm SINGLE
var
  tjoin : timer
  single_id : view_id
  joined : list of nodes
  joining : list of nodes
  tmerge : timer
begin
   tjoin := 0;
   tmerge := -1;
   if first startup then
     single_id := zero_id;
   else
     single_id := singleton view_id
       from majority reference;
   fi;
   if current.size()!=1 then
     current := (single_id,
       thisNode, thisNode);
     weak.add((current,COMMITTED));
     commit(current);
     joined := empty;
     next := null;
   fi;
   wait for event
   case event of
   recv Join(V):
     if thisNode leads then
       joining.add(V.mbrs);
       multicast( Joined(current.id,
         V.id, majHist-V, lastPrep) )
         to V.mbrs;
     else if lastPrep>V or
      majHistory>V then
       send( Update(majHist-V,
        lastPrep) ) to sender;
     fi;
   recv Update(views, Vmaj):
     for V in views-majHistory do
       upcommit(V);
     done;
     lastPrep := Vmaj;
   recv Ready:
     joined.add(sender);
     joining.remove(sender);
     if joining is empty then
       tmerge:=0;
     fi;
   recv Joined(V,current.id):
     send(Ready(V)) to sender;
     state:=MERGE;
   tjoin timeout:
     broadcast(Join(current,majHistory.
      last(),lastPrep));
     tjoin:=TJOIN;
   tmerge timeout:
     next := (next_id, thisNode,
       current U joined);
     if next is majority then
       state:=majMSTR;
     else if joined not empty then
       state:=minMSTR;
     fi;
   esac
end
```

Figure 4.3: Algorithm for the SINGLE state of HMS.

```
algorithm majMSTR
var
  step : integer
  pending : list of nodes
  joining : list of nodes
  joined : list of nodes
  tmerge : timer
init
  step := 0
  pending := empty
  joining := empty
  joined := empty
  tmerge := -1
begin
  if next!=null and pending is empty then
    next.setid(next majority id);
    if strong.lastprepared() not committed then
      current := strong.lastprepared();
      strong.set(current,COMMITTED);
      commit(current);
    fi;
    strong.add((next,PREPARED));
    prepare(next);
    failed := empty;
    pending := next.mbrs;
    rmcast(Setmem(next, strong.lastcommitted())) to next.mbrs;
    step := 0;
    joining := empty;
    joined := empty;
    tmerge := -1;
  fi;
  wait for event
  case event of
  recv Ends(V,n):
    if n<2 then
      pending.remove(sender);
      if pending is empty then
        step := step+1;
        if step==1 then
          strong.set(V,COMMITTED);
          commit(V);
          maHistory.add(V);
          current := V;
          next := null;
          pending := current.mbrs;
        else if step==2 then
          strong.release(current);
          release(current);
        fi;
        rmcast(Step(current,step)) to current.mbrs;
      fi;
    fi;
  recv Join(V, lastV):
    if step>=2 then
      joining.add(V.mbrs);
      multicast(Joined(curent.id,V.id majHistory-lastV)) to V.mbrs;
      tmerge := TMERGE;
    fi;
  recv Ready(current):
    joined.add(sender);
    joining.remove(sender);
    if joining is empty then
      tmerge := 0;
  fail(mbrs):
    failed.add(mbrs);
    discard(strong.lastprepared());
    strong.remove(lastprepared());
    next := current.mbrs U joined.mbrs \ failed;
    joining := joined := empty;
    pending := null;
  tmerge timeout:
    next := current.mbrs U joined.mbrs;
    joined := empty;
    tmerge := -1;
esac;
end
```

Figure 4.4: Algorithm for the majMSTR state.

```
algorithm majMBR
var
  step :  integer
init
  step := 0
begin
  if next!=null then
    step := 0;
    if nextC!=null then
      strong.confirm(nextC);
    fi;
    strong.add((next,PREPARED));
    rsend(Ends(next.id, 0))
     to next.master;
    next := nextC := null;
    failed := empty;
    substit := null;
  fi;
  wait for event
  case event of
  recv Setmem(V,Vc):
    if V is majority then
      step = 0;
      strong.commit(Vc);
      strong.add((V,PREPARED));
      failed := empty;
      majHistory.add(Vc);
      lastPrep := V;
      rsend(Ends(V.id, 0))
       to V.master;
    else
      next := V;
      state := minMBR;
    fi;
  recv Step(V.id,1):
    step := 1;
    current := strong.last();
    strong.commit(current);
    rsend(Ends(V.id, 1))
     to V.master;
  recv Step(V.id,2):
    step := 2;
    strong.release(current);
    release(current);
  recv ChangeM(V):
    if V.master==current.master then
      failed.add(current.master);
      rsend(View(strong.lastprepared(),
       strong.lastcommitted())
       to sender;
      if !(V<last committed) then
        substit := sender;
      fi;
      state := mstrCHG;
    fi;
  fail(mbrs):
    failed.add(mbrs);
    if thisNode is master of
     current.mbrs \ mbrs then
      next := next.remove(mbrs);
      state := SUBST;
    fi;
esac
end
```

Figure 4.5: Algorithm for the majMBR state.

```
algorithm minMSTR
var
  step : integer
  pending : list of nodes
  joining : list of nodes
  joined : list of nodes
  tmerge : timer
  tjoin : timer
init
  step := 0
  pending := empty
  joining := empty
  joined := empty
  tmerge := -1
  tjoin := -1
begin
  if next!=null and
   pending is empty then
    next.setid(next minority id);
    if weak.lastprepared() not
     committed then
      current := weak.
      lastprepared();
      weak.set(current,COMMITTED);
      commit(current);
    fi;
    weak.add((next,PREPARED));
    prepare(next);
    failed := empty;
    pending := next.mbrs;
    rmcast(Setmem(next, weak.
     lastcommitted(), lastPrep))
     to next.mbrs;
    step := 0;
    joining := joined := empty;
    tmerge := -1;
    tjoin := -1;
  fi;
  wait for event
  case event of
  recv Ends(V,n):
    as in majMSTR with strong -> weak
    if pending is empty and
     step==2 then tjoin := 0;
  recv Join(V,lastV):
    if step>=2 then
      if thisNode leads then
       joining.add(V.mbrs);
       multicast(Joined(curent.id,
        V.id,majHistory-lastV))
        to V.mbrs;
       tmerge := TMERGE;
      else if lastPrep>V or
       majHistory>V then
       send(Update(majHist-V,
        lastPrep)) to sender;
      fi;
    fi;
  recv Ready(current):
  fail(mbrs):
  tmerge timeout:
    as in majMSTR with strong -> weak
    and explicit checking of majority
    histories freshness for merging
  recv Joined(V):
    send(Ready(V)) to sender;
    state := MERGE;
  recv Update(views, lastP):
    if step==2 then
      rmcast(Update(views,
       lastP)) to current.mbrs;
      pending := current.mbrs;
      state := UPDAT;
    fi;
  tjoin timeout:
    broadcast(Join(current,majHist.
     last(),lastPrep));
    tjoin := TJOIN;
  esac;
end
```

Figure 4.6: Algorithm for the minMSTR state.

```
algorithm minMBR
var
  step :  integer
init
  step := 0
begin
  if next!=null then
    step := 0;
    weak.add((next, PREPARED));
     rsend(Ends(next.id, 0))
      to next.master;
     next := null;
   fi;
   wait for event
   case event of
   recv Setmem(V, Vc):
     if V is majority then
       update majHistory with Vc;
       next := V;
       state := majMBR;
     else
       step := 0;
       weak.commit(Vc);
       weak.add((V, PREPARED));
       failed := empty;
       rsend(Ends(V.id, 0))
        to V.master;
     fi;
   recv Step(V.id, n):
   recv ChangeM(V):
     as in majMBR with strong -> weak and
     checking consistency of majHistory for
     ChangeM
   recv Joined(V,Vid):
     if Vid=current.id and
      step>=1 then
       send(Ready(V.id)) to sender;
       state := MERGE;
   recv Update(views, lastP):
     update majHistory with views;
     strong.add((lastP, PREPARED));
     lastPrep := lastP;
     rsend(Ends(4)) to mstr;
end
```

Figure 4.7: Algorithm for the minMBR state of HMS.

```
algorithm SUBST
var
  tsubst :  timer
  answers :  list of View msgs
  failed :  list of nodes
init
  tsubst := -1
  answers := empty
  failed := empty
begin
  substit := thisNode;
  tsubst := TSUBST;
  pending := current.mbrs U
    next.mbrs \ failed;
  if current is minority
   then
    multicast(ChangeM(next, last
     committed)) to pending;
  else
    multicast(ChangeM(next, last
     committed, lastPrep))
     to pending;
  fi;
  tsubst := TSUBST;
  answers := View(next, last
   committed) from thisNode;
  failed := empty;
  wait for event
  case event of
  recv View(Vp, Vc):
    if Vc>next or (next==null and
     Vc>last committed) then
      pending := null;
      state := SINGLE;
    fi;
    answers.add(msg);
    pending.remove(sender);
    if pending is empty then
      analyse(answers, failed)[a];
    fi;
  fail(mbrs):
    pending.remove(mbrs);
    failed.add(mbrs);
    if pending is empty then
      analyse(answers, failed);
    fi;
  tsubst timeout:
    analyse(answers, failed);
esac
end
```

[a]See Fig. 4.9

Figure 4.8: Algorithm for the SUBST state.

```
procedure ANALYSE(answers,failed)
  process all received answers to decide on confirmation of views (and preparation of last
  majority view, in case of minority operation)
  if thisNode is excluded then
    state := SINGLE;
  else
    if answers contains (next,COMMITTED) then
      set(next,COMMITTED) in strong or weak;
      commit(next);
    fi;
    next:=answers.senders() \ failed;
    if next is majority then
      state := majMSTR;
    else
      set lastPrep if required;
      state := minMSTR;
    fi;
  fi;
```

Figure 4.9: Procedure for analysing answers to master change proposal.

```
algorithm mstrCHG
var
  tchg :  timer
init
  tchg := -1
begin
  if thisNode is next substitute them
    tchg :=TCHG;
  fi;
   wait for event
   case event of
   recv ChangeM(V):
     if V.master==(current.master or
       substit) and sender is a
       valid substitute then
       if !(V<last committed) then
        failed.add(substitutes until;
          sender);
        substit := sender;
        send(View(last committed,
          last prepared)) to sender;
       fi;
     fi;
   recv Setmem(V,Vc,lastP):
     next := V;
     if Vc minority then
       weak.set(Vc,COMMITTED);
       commit(Vc);
       current := Vc;
     else if msg contains
      (Vc,COMMITTED) then
       strong.set(Vc,COMMITTED);
       commit(Vc);
       current := Vc;
     else;
       upcommit(Vc);
     fi;
     confirm Vc if required;
     if V is minority then
       if weak contains V'<V prepared
         then
         weak.remove(V');
         discard(V');
       fi;
       strong.add((lastP,PREPARED));
       lastPrep := lastP;
       state := minMBR;
     else
       if strong contains V'<V
         prepared then
         strong.remove(V');
         discard(V');
       fi;
       state := majMBR;
     fi;
   fail(mbrs):
     failed.add(mbrs);
     if failed contains all
      substitutes then
       state := SUBST;
     fi;
   tchg timeout:
     failed.add(substit);
     state := SUBST;
   esac
end
```

Figure 4.10: Algorithm for the mstrCHG state.

```
algorithm UPDAT
begin
  wait for event
  case event of
  recv Ends(4):
    pending.remove(sender);
    if pending is empty then
      state := minMSTR;
    fi;
  fail(mbrs):
    pending.remove(mbrs);
    if next==null then
      next := current.mbrs \ mbrs;
    else
      next := next \ mbrs;
    fi;
    if pending is empty then
      state := minMSTR;
    fi;
  esac
end
```

```
algorithm MERGE
begin
  twait := TWAIT;
  wait for event
  case event of
  recv Setmem(V,Vc,lastP):
    lastPrep := lastP;
    if Vc is majority then
      majHistory.add(Vc);
      upcommit(Vc);
    fi;
      next := V;
    if V is majority then
      state := majMBR;
    else
      state := minMBR;
    fi;
  fail(M):
    state := SINGLE;
  twait timeout:
    state := SINGLE;
  esac
end
```

Figure 4.11: Algorithms of UPDAT and MERGE states.

4.3.2 Basic Operation

The basic algorithm for view proposal and confirmation is similar to the S-GMP protocol proposed in [8] for the Isis system, that was described in sect. 2.2.1. The S-GMP protocol was basically a two–phase commit led by a manager, and a three–phase algorithm in case of manager failure. The HMS algorithm includes an extra phase and the management of stable storage to ensure full recoverability in case of multiple failures. Besides, HMS ensures operation also in minority partitions, even when the majority is lost, and the remerging of groups to recover the majority.

Initial View

HMS uses individual startup. Each node starts in the SINGLE state of fig. 4.3, installing a predefined *zero* configuration, where it is the master and the only member. The identifier of the *zero* view is $Vid(V_0) = (0, N_{\mathrm{m}}, 0)$, being N_{m} the static identifier of the node, and the composition of the group is $\{N_{\mathrm{m}}\}$.

If the node is not starting for the first time, but recovering after a crash, then the stable storage keeps the last installed majority view, if it exists, $(M, -1, -1)$, and the last number of mastership incarnation used by N_m, k. In that case, the node composes and installs a view with the same composition of the *zero* view and a new identifier $(M, N_m + (k + 1) \times N, 0)$, and it correspondingly increases the stored value k.

Being in minority, the master N_m immediately starts broadcasting **Join** messages trying to locate and merge with other existing partitions. If the master of an existing external group receives **Join** from N_m, the *Partition Joining Protocol* described below will allow merging of both parts.

View Proposal and Confirmation

The master node takes charge of proposing new views, deciding whether they must be confirmed or discarded, and announcing these decissions to the rest of nodes. At some point after the master has proposed a view, and if no more failures occur for a long enough period of time, the configuration that is being installed can be promoted to the definitive category, saved to stable storage and notified to higher level applications that may be making use of membership services. Our protocol ensures that once this step has been taken by any single node, all the living members of the group will also perform it. The basic operation of the protocol, described in this section, guarantees that consistency is kept among view histories, regardless of message losses or delays, provided the majority character of the partition is preserved by changes.[2] The required

[2]The special case of master failure will be separately dealt with in section 4.3.2.

operations correspond to the states majMSTR and majMBR, described in figs. 4.4, 4.5 respectively.

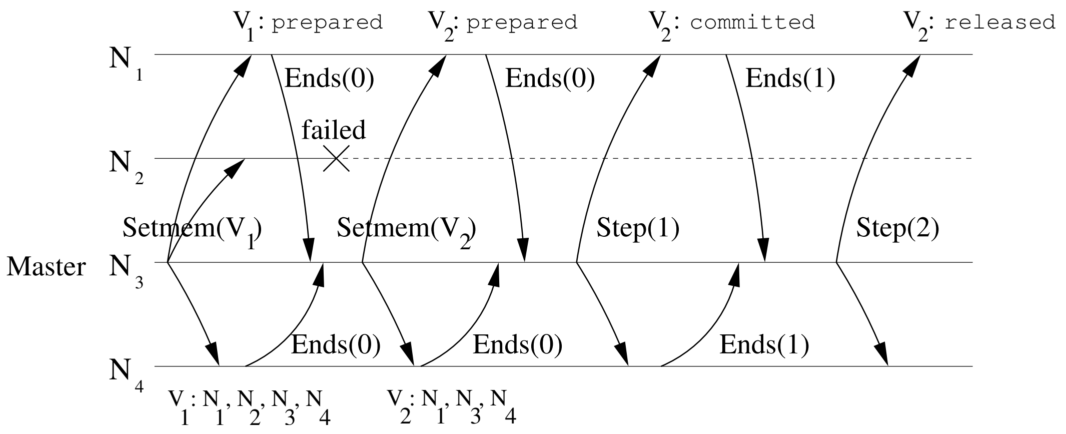


Figure 4.12: Proposal and confirmation of a view in the HMS protocol, with one failure.

To achieve the desired guarantees, the confirmation of a view is carried out as a three-phase commit, as depicted in fig. 4.12.

1. When the master is aware of a change in the membership set, it composes a new view V_p to be proposed, by adding or removing members from the last entry in its *strong* list. Then V_p is added to the *strong* list in the following way.

   (a) The list is checked backward to identify the last `committed` view, V_c.

   (b) Any view in the list with status `prepared` is discarded.

   (c) The pair (V_p, `prepared`) is appended to the end of the list.

   Next the master broadcasts the message **Setmem**(V_p,(V_c, confirm)) to all members in V_p.

2. When another node receives that message, it processes its contents and updates its own *strong* list accordingly.

   (a) If V_c was `prepared` in the list, it is now promoted to `committed`.

   (b) If the list contained any previous view in the `committed` state, that is now promoted to `released`.[3]

   (c) (V_p, `prepared`) is appended to the list, after discarding any previously `prepared` view.

[3]The same checking for older `committed` views on the list is performed each time a node promotes a view to this status.

After processing the **Setmem** message, the node will acknowledge the reception of the view by sending an **Ends**(0) message to the master. The integer argument of the message (0 in this case) indicates the number of the completed phase.

3. The master may start the next phase of the protocol when it has received all the **Ends** messages for phase 0. It then locally installs V_p, labelling it as `committed` in the *strong* list and broadcasts a **Step**(1) message, which represents an order to install the view for the remaining nodes. Locally labelling a view as `committed` implies that the view has been confirmed. Therefore at this point higher level applications that make use of the membership service are informed about the irrevocable installation of V_p. At the same time, information on this view is saved to the *majority history*. If, before the master has received all the **Ends**(0) messages, some change is detected (e.g. from a suspect of the failure detector), the master will not install V_p, but go back to the first stage of the protocol, to compose a new view V_p' and start over. Conversely, if joining of some node has been noticed in the meanwhile, the master does not restart the protocol immediately to propose an enlarged view, but it waits until V_c is ready to be committed (unless a failure occurs), so that the **Setmem** message will serve to spread confirmation of V_p.

4. When another node receives the **Step**(1) message, the configuration is saved, and also labelled as `committed`. Once the **Step** message has been processed in such way, the node replies with an **Ends**(1) message to the master.

5. After receiving all the **Ends**(1) messages, the phase 1 is concluded. Then the Master labels V_p as `released` and broadcasts **Step**(2). After receiving the last message the rest of nodes can also promote the view to `released`, since no later reconfiguration due to master change will need this information.

Since `committed` views are automatically released when confirming later views, the extra `release` phase is not required when changes follow one another with short interval. This extra phase then has no cost during periods of unstability, in which it is not advisable to inject extra messages in the system. When a view becomes stable, instead, it allows each node to release their resources, as the information on the `released` view will not be needed by subsequent operations, and to inform upper components that the installation has been completed at all nodes.

Master Change

When the master fails, some other node has to take its role. The identity of the substitute is deterministically decided, as that of the master. In our implementation [20] the survivor with the lowest identifier is assumed to take the master's place, but other

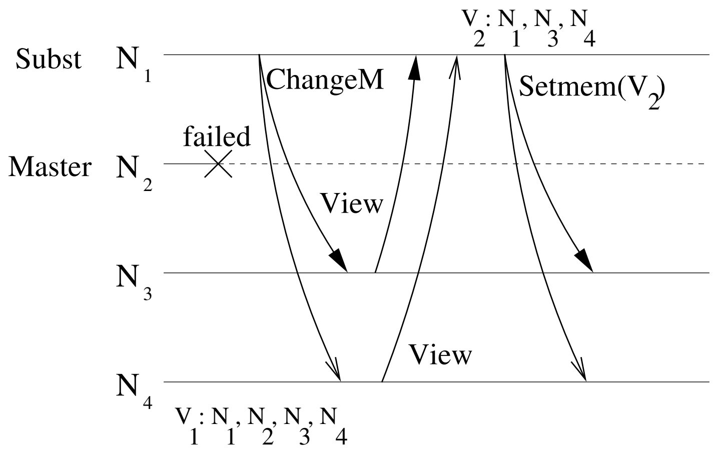


Figure 4.13: Master changing in the HMS protocol.

criteria (such as the longest living node) would be equally suitable, as long as the same deterministic criterion is used by all nodes.

The failure will be eventually detected by the failure detector, since it satisfies the property of *strong completeness* (FD.1) and notified to every correct node. In particular, then, the substitute will notice the failure of the master node, move to the SUBST state and initiate the master changing procedure described in fig. 4.10 and illustrated in fig. 4.13.

An additional protocol phase is required to overcome the master's failure. When the substitute master, N_{PM}, notices the failure it must collect some information from the rest of nodes in the group before proposing a view that excludes the old master. The would–be master enters the SUBST state. It will then check its own *strong* list to figure out whether there was a view in process of being installed. Next it sends a **ChangeM** message including the last entry of the list, say V_ℓ, to all the nodes that may be part of the group. This includes all nodes in V_ℓ and nodes from the last `committed` view, if different from V_ℓ, as they could have been unfoundedly excluded.

When a node receives the message **ChangeM**(V_ℓ), it moves to the mstrCHG state described in fig. 4.10, and composes a **View** message in answer, according to the following rules.

1. First, the local *strong* list is checked, and the last locally `committed` and `prepared` views, respectively V_c and V_p, are identified. If the *strong list* does not contain any `committed` view, as might be the case if the last one was already `released` or if the node did just join the majority group, then the last element of the *majority history* is sent instead.

2. If ($V_c > V_\ell$), then N_{PM} does not longer belong to the group, as it was excluded

from the already committed V_c. Then the **ChangeM** message from N_{PM} is not acceptable, as a different substitute will propose itself as master within the confirmed group. Anyway, a **View**(V_c, committed) may be sent in answer to N_{PM} to notify it about its exclusion.

3. If ($V_c = V_\ell$) and ($V_p > V_\ell$), then this node has received a proposal later than V_ℓ. If ($N_{PM} \in V_p$), **View**(V_ℓ, committed)) is sent to N_{PM}. Otherwise, **View**((V_ℓ, committed), (V_p, prepared)) is sent.

4. If ($V_c < V_\ell$) the answer to N_{PM} is **View**(V_p, prepared).

The would–be master N_{PM} waits until either all the replies from live destinations have been received or a proper timeout has expired (to account for the possible dismissal of the **ChangeM** message by nodes in a confirmed view) and analyses all answers.

1. If any committed $V_c > V_\ell$ has been reported, then necessarily $N_{PM} \notin V_c$, so that N_{PM} excludes itself from the group, enters state SINGLE and installs a singleton view using as reference the first component of the last confirmed majority view identifier.

2. If a prepared $V_p > V_\ell$ is noticed, such that $N_{PM} \notin V_p$, but it is not explicitly declared as committed by any node, then N_{PM} will only be excluded if all nodes in $V_p \cap V_\ell$ report V_p as prepared. This is so because if all the surviving participants have prepared V_p, it is possible that the old master received all answers and committed the view before failing. In that case, uniformity requires that the view is also confirmed by the surviving nodes. Since it is not possible to know whether the old Master did commit or not V_p, the surviving group must always confirm it if the possibility of commitment exists. Instead, if some node participating in such V_p did not prepare it at all, the former master could not possibly confirm the view. It is then safe to discard V_p, so that N_{PM} prepares the new view to be proposed, V_n and sends a **Setmem**(V_n) message to its members. If some other node had locally committed V_ℓ (information that can be deduced from the received **View** messages), the **Setmem** message will also contain the order to confirm it. Otherwise V_ℓ must also be discarded, since the old Master proposed a later view without confirming it.

3. Finally, if none of the above conditions holds, then either all views reported by the rest of nodes are lower or equal to V_ℓ, or any reported $V_p > V_\ell$ is such that $N_{PM} \in V_p$. such a V_p can be safely discarded, since at least one member of the view, N_{PM}, did not receive the proposal and could not confirm it. In such case, when N_{PM} prepares the new view V_n and sends the corresponding **Setmem(**V_n**)** message, if all nodes in both views have reported V_ℓ as prepared then the message includes the confirmation of V_ℓ with a special label to indicate it has to be `upcommitted` (as it is not a real installation).

An acceptable **ChangeM** message (i.e. one coming from a member of the last known group, that has not been confirmedly excluded before the master's failure) has the effect to block reception of further messages from the old master. When the rest of nodes receive an acceptable **Setmem** message from the substitute, they will enter normal operation in majMBR state. Nevertheless, a timeout should be established, so that a second node is able to enter the SUBST state and assume the mastership in case the first candidate ends up by excluding itself.

If the resulting group V_n is the majority, confirmation of the last view V_ℓ has a definitive character, and any failed (or partitioned) node that is contained in the view will have to be notified about this configuration whenever it tries to join again the majority group. In minority operation, some additional considerations are required, which we describe in sect. 4.3.3.

4.3.3 Minority Operation

One distinguishing feature of the HMS protocol is its partitionable character, that allows full operation in minority mode and guarantees the consistency of the majority history when partitions are merged. This section describes in detail all those actions and special considerations involved by operation in the minority mode. The majority condition may be lost when the cardinality of the group is reduced by failures or partitions. The minority operation affects view proposal and master changing procedures, but also merging of partitions which always involves at least one minority group. A particular situation arises when two minority partitions are to be merged. This case deserves specific attention and will be separately discussed later in this section.

Minority View Proposal and Confirmation

Since in an asynchronous system node failures are indistinguishable from network partitions, when an exclusion causes the group to lose the majority condition, confirmation or discarding of previous `prepared` majority views must be postponed, as there might exist a disjoint majority group which has taken a different resolution on such views. Thus the algorithm described in sect. 4.3.2 for proposing and confirming views must be slightly modified in case the group is in minority. Such differences are made explicit in the algorithms for minMSTR and minMBR states, pictured in figs. 4.6, 4.7.

If the master has prepared a view V_n which turns out to be in minority, before sending the **Setmem** message, it switches the view list. The *strong list* is set aside without modification, and the *weak* list is initialised with the entry (V_n, `prepared`). From that moment, the master will handle the *weak* list instead of the *strong* one, until the majority status is restored. The **Setmem** message in this case will contain not only the just prepared minority view but also the latest confirmed and prepared majority views, in order to ensure a uniform knowledge about the majority across the group for easing a future merging with a majority. When another node receives the minority proposal from the Master, it will perform the update of the *strong* list and then an identical switch to the *weak* list. Proposal, confirmation or discarding of views in the minority condition, using the *weak* list, is done in the regular way described in sect. 4.3.2, but `committed` minority views do not have the same definitive character as majority ones, and are not stored in the stable history.

Moreover, higher level applications that make use of HMS membership service are informed about the fact that the group is not the majority, so that their actions derived from these membership changes respect whatever application–dependent restrictions may be required for minority operation.

Master Change

The master changing protocol must also take into account the fact that the group stops being a majority in order to guarantee the desired behaviour.

Once the master of a majority group fails and its substitute N_{PM} initiates the master changing protocol, it can realise that the left group is no longer a majority, or that it turns out to be excluded by answers of the remaining members thus becoming a singleton. The latter case is easily solved by installing a singleton view. The former case, instead, requires that all participants enter the minority mode while preserving the consistency properties of the majority history. Therefore, once the substitute N_{PM}, has checked that none of the exclusion conditions holds (so that it does not have to exclude itself), some care has to be taken when analysing the answers of surviving nodes and proposing the new view.

1. The confirmation of V_ℓ (by including (V_ℓ, confirm) in the **Setmem** message) is possible only if it appears explicitly as `committed` in some list of the surviving nodes. However, if it is simply `prepared` in all its surviving participants, no `upcommit` takes place.

2. A (V_p, `prepared`) view found in **View** messages such that $V_p > V_\ell$ cannot be discarded even if not every node in V_p has reported it as `prepared`, regardless of whether $N_{PM} \in V_p$ or $N_{PM} \notin V_p$. It might be the case that a majority partition existed in which all V_p participants had it `prepared`, so that the majority master took the decision to confirm it.

3. Nevertheless, a `prepared` view V_p can be discarded if a later `prepared` view V_p' has been reported. If V_p had been confirmed by the majority master, any node receiving a later view V_p' should have it confirmed, either in its strong list or in the majority history. But if that were the case N_{PM} would have seen that

confirmation in the arguments of the **View** messages. Therefore the new master will identify the most recent among all the `prepared` majority views, say V_P, and include the pair (V_P, prepare) in the **Setmem** message.

When processing the **Setmem** message each node will install the new proposed view in the *weak* list and update its *strong* list as the infomation in the message indicates. If the node held a `prepared` view, it will now label it `committed` and produce a `commit` event only if the Master included the corresponding confirm instruction in the message. Otherwise, the view will be discarded and substituted by the more recent view the master announces now as `prepared`. If no `prepared` view was present in the list, it may now add one according to the master's instructions. Afterwards, the node will operate on the *weak* list while the *strong* one is left pending until the node tries to join a majority group.

The above description applies to the start of minority operation, when the majority character is lost but the operation of master change within minority is basically the same.

Partition Merging

The process of partition merging is fundamental for partitionable operation. It includes those actions required to reincorporate a minority group to the majority, but also to unite two minority groups into a larger, still minority view in a consistent way, and to compose a majority by merging several minority groups whenever the majority has disappeared from the system. This section deals with the first case, while the second and the third require some specific actions that will be described in detail in the next sections.

In HMS, this process is the same employed by new nodes that wish to join the group for the first time, since a new node will install a singleton view at startup and will thus necessarily be in a minority group at the beginning.

Any minority master, N_{B}, in a `released` view, V_{B}, will periodically broadcast a **Join** message to probe for a larger group. **Join** messages include the composition of the current view and the identifier of the last majority view it knows about, say V_{M}. Such a message can be attended by a majority master, N_{A}, leading a stable configuration V_{A} (i.e. a `released` view, such that all the participants have acknowledged its confirmation), that may start the partition merging protocol and allow nodes in the smaller group to enter the majority partition.

Merging is only allowed between partitions with a common *majority history* prefix, except for the last `prepared` view in each *strong* list. Thus when a majority Master gets a **Join(**V_{B}**,** V_{M}**)** message, it will check which part of the *majority history* is missing in the minority group, by comparing the last majority view reported by the **Join** message with its own *history*. Then it will send a **Joined(**$Vid(V_A)$**,** $Vid(V_B)$**,** $V_M - V_A$**)** message to each member in the minority group. The first argument of the message, $Vid(V_A)$, contains its own view identifier, whereas $V_M - V_A$ represents the part of the *majority history* unknown to the join proposer.

When receiving the **Joined** message, each node in the minority group, if it is in a stable state (not in process of view preparation), will check whether the reference minority view identifier in the message, $Vid(V_B)$, agrees with its current view. If that is the case, the node will enter the MERGE state of fig. 4.11, individually update its *strong* list if possible, and reply to the majority Master with a **Ready** message containing the received majority view identifier so to allow N_A to discard obsolete answers.

The majority master will collect **Ready** messages and add their senders to the new view proposal. The wait ends when all the expected **Ready** have been received, or a failure occurs within V_A forcing a new view, or a certain time period elapses. The latter condition is intended to avoid N_A blocking in an indefinite wait in case there have been changes to the minority group that prevent nodes from continuing with the partition merging protocol. When the wait finishes the majority master proposes the enlarged view with a **Setmem** message.

If changes have happened to the majority group between the sending of **Joined** and the arrival of **Ready** messages, the majority view identifier will have changed and the reference used by the minority nodes will not longer be valid. In this case, joining has to be delayed until both groups share the same knowledge about majority history. Therefore, a new **Joined** message may be sent with the last changes and the new reference identifier, and the joining can be retried.

When receiving the **Joined** message from N_A each minority member abandons the group led by N_B, so that consistency is kept among majority histories of nodes belonging to the same view, and waits for a view proposal from the majority master. After the exchange of the corresponding **Ready** messages and the arrival of the **Setmem** from N_A these nodes will enter the majMBR state, take the message sender as their new master, and join the majority group. A timeout is also established so to end this wait after a certain period, in order to avoid a blocking if a certain node got excluded from the new view to be formed. The wait will also be interrupted if N_A is detected to have failed. In such cases, having abandoned the group V_B, each node will install the corresponding singleton view and restart sending periodic **Join** messages.

This procedure applies to the case in which the **Join** request from the minority reaches a majority partition. But merging is also possible between two minority groups in order to form a larger one, if several minority groups exist. In particular, such merging will be definitely needed if the system has split and lost the majority at all, to recover it. In the case of merging two minority partitions, the largest group, is in charge of leading the partition merging protocol, acting as the majority master in the procedure described above. If both minority partitions have the same cardinality, the one that holds the would–be master of the joint set, according to the deterministic master selecting rule, will lead the merging. The rest of participants will enter the minMBR state when receiving a **Setmem** message that contains the proposal of a minority view.

View History Matching

When two minority partitions exist, they may get in touch and attempt joining. Unlike in the case of merging led by the majority, the resulting master of the joint group from two minorities does not necessarily have the most up to date information on the majority history. This raises the need for a special procedure to share that information between both parts before running the partition merging protocol. That goal is achieved by the *view history matching* subprotocol described in this section.

When a minority master N_{m_1} gets a **Join** message carrying an outdated version of the majority history, and provided it is not in the situation to assume the leadership to merge both partitions, it will reply to the sender with a special **Update** message containing the part of the majority history the other group is missing. This includes also the last prepared majority view that may have been left pending after the partition.[4] The minority Master receiving such **Update** message will refresh its majority history and propagate this information to the rest of its group. The updating group will not accept other **Join** messages until all *strong* lists agree (see algorithm of the UPDAT state in fig. 4.11). When updating their histories, nodes may find that some `prepared` view they had suspended in the *strong* list has been confirmed by the majority. Therefore, they will proceed accordingly, labelling it `committed` and then `released`. Conversely, if the `prepared` view is not present in the majority history updates, they will discard the view.

A special situation arises when joining of two minority partitions adds up to a majority group. The subprotocol for view history matching and the partition merging procedure ensure that, at the point of sending the majority **Setmem**, the master knows that all participating members have a common, up-to-date knowledge of the majority history, including all confirmed majority views and perhaps a last `prepared` view. Among the possibly multiple `prepared` majority views that can be left pending in different minority partitions when the majority is lost, all merging protocols ensure that only the most recent one is kept while the others are either confirmed, if evidence exist for their installation, or discarded. Such latest `prepared` majority view, say V_p must now be confirmed. Thus the master will propose the new majority view V_n by sending a **Setmem**(V_n,(V_p,confirm)) message.

Nodes accepting the **Setmem** will process their remaining *strong* list according to

[4]Such `prepared` view must also be included in the **Joined** message in case the most up-to-date partition is leading the merging process.

the information on V_{p} sent by the new Master, and then append V_n to that list as `prepared`, while the *weak* list is discarded.

4.4 Performance of HMS

The HMS protocol has been fully implemented in Java as a membership monitor that provides the generic interface in 3.1. The protocol makes use of a external failure detector, and a basic layer providing unreliable transport. All components are implemented in the context of HAMS project, so that communication among them takes place by means of events, and they fulfil the generic interfaces of notifiers and listeners defined therein. We have run a series of tests to check the correctness of the implementation and also to characterise the protocol performance.

The behaviour of the protocol depends on a large number of parameters. Besides, the uncountable scenarios and situations that may arise during the execution multiplies the observable quantities capable to be used as performance measurements. This is probably the reason why no standard metrics exists for membership protocols.

In order to characterise the performance of HMS, then, the following time quantities have been defined.

- T_{join} is the time elapsed from the moment the master receives a **Join** message until a new view is installed.
- T_{fail} is the time from the reception of a failure notification from the failure detector until a new view is installed.
- T_{comm} is the time elapsed from the installation of the initial *zero* view when a node starts up until it installs the next view, i.e. it is integrated in the group.

These quantities give an idea of the time cost of the fundamental protocol operations. In particular, T_{join} and T_{comm} measure the time of reaction to the incorporation of new members, the former from the point of view of the already existing group and the latter from the point of view of the new node. Conversely, T_{fail} measures the time cost of excluding a node that has been judged failed by the failure detector.

The absence of a global time in asynchronous distributed systems forces these quantities to be defined in terms of local clocks. Therefore, each of them is defined as a difference between the value of a node's local clock when the installation of a view takes place minus the corresponding value when the former event (reception of **Join**, failure notification or installation of *zero* view) took place. In our experimental setting, all nodes are launched in identical processors, and thus we may safely assume that all clocks have a comparable speed, and thus we average the measured quantities over all nodes in the system. This implies no loss of generality, since in case of different clock drifts, the measurements could equally be performed locally and the speed of the clocks compared to an external reference so that the results can be expressed in terms of a single time unit.

Given the large number of parameters involved in the protocol, and the multitude of possible situations, a reduced set of scenarios was selected in order to perform the characterising measurements. In particular, we measured the quantities defined above in situations when the change in the group involved just one node.

Since one of the major distinguishing features of HMS is the different handling of majority and minority groups, it is necessary to measure the time costs in both operation modes. Besides, changing between both modes requires specific actions to be taken, so that the time cost of joining or leaving the group may be different if the group change implies a change in the mode of operation. To evaluate the impact of HMS majority handling on the time cost of joining and leaving the group, the measurements were also performed on scenarios in which each connection or disconnection implied moving from minority to majority and vice versa.

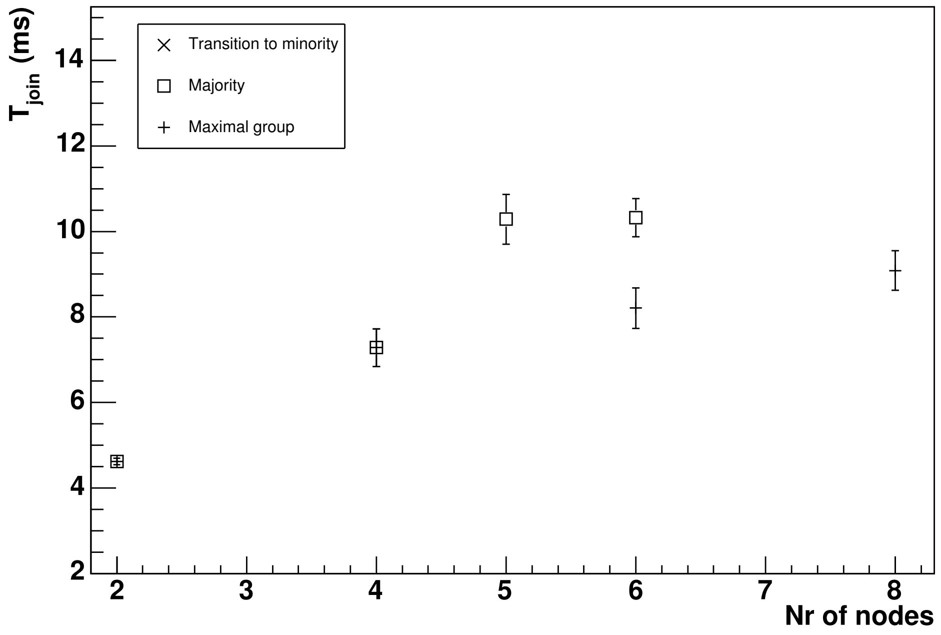


Figure 4.14: Mean value of T_{join} as a function of the group size.

For each experiment, a size N was chosen for the preconfigured cluster, and a total number of nodes, $m \leq N$, was launched. Then join and leave operations were forced by successively disconnecting and connecting a single node, by turns. For each size of

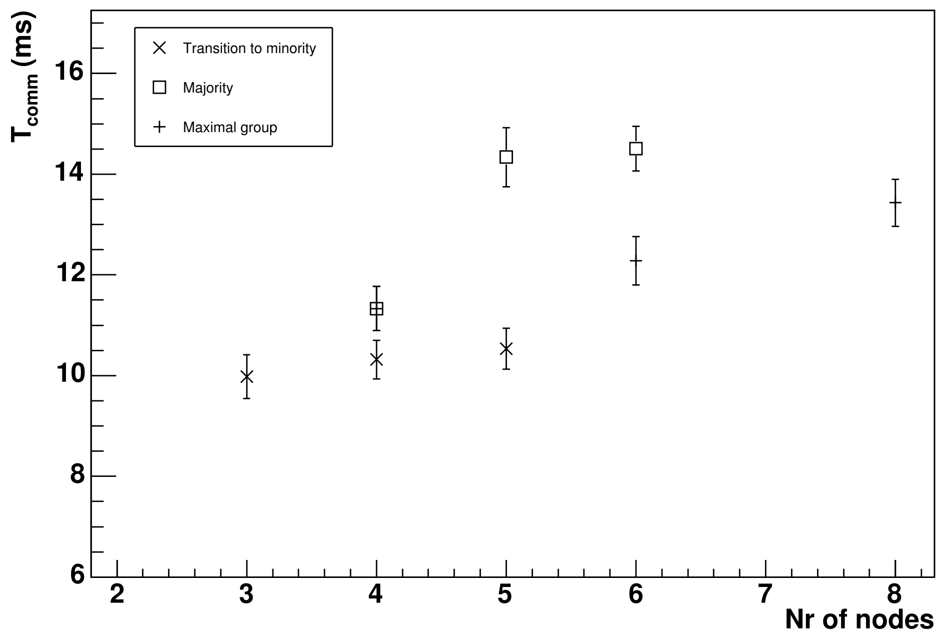


Figure 4.15: Mean value of T_{comm} as a function of the group size.

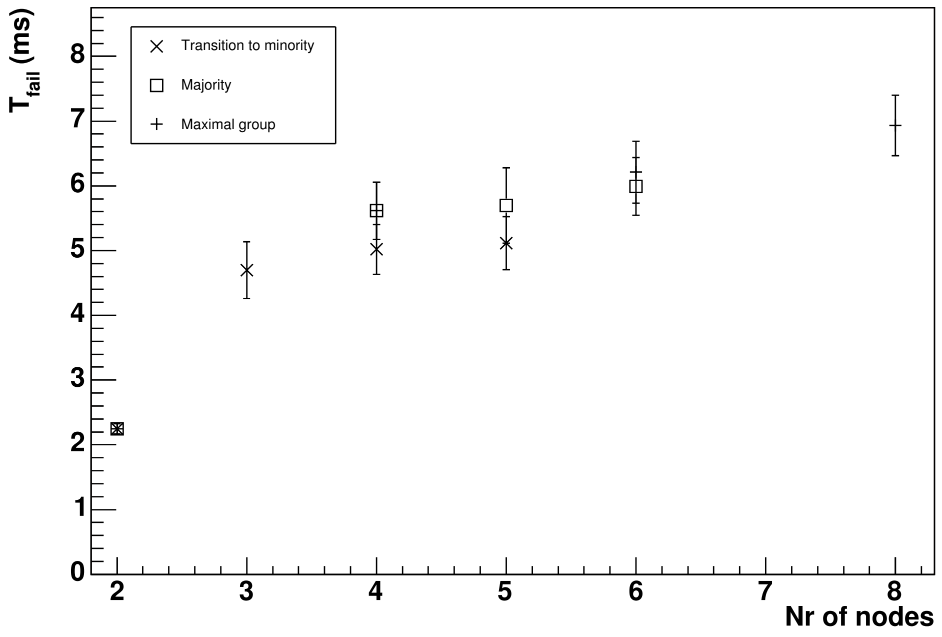


Figure 4.16: Mean value of T_{fail} as a function of the group size.

the cluster, various values of m were tested, in order to reproduce the different settings discussed above.

- $m = \frac{N}{2} - 1$ is a strictly minority. Successive disconnection and rejoining of a single node makes the size of the group vary between $\frac{N}{2} - 2$ and $\frac{N}{2} - 1$, so that the majority is never reached and the cost of the fundamental operations in minority can be evaluated.

- $m = \frac{N}{2} + 1$ represents a majority group, but a single disconnection makes the size vary between $\frac{N}{2}$ and $\frac{N}{2} + 1$, so that each change implies a change of mode to or from minority. This setting thus allows us evaluate the cost of operation mode changes.

- $m = \frac{N}{2} + 2$, $m = N$ define groups in strictly majority, also after a single node disconnects, so that the observables can be measured in majority operation.

A first set of experiments was carried on a cluster of 8 PCs (Intel® Pentium® 4, CPU 2.80 GHz). The processors were running the Linux kernel (version 2.4.22) and JDK 1.4. We chose a total size of the system $N \in \{2,\ 4,\ 6,\ 8\}$. With such realistic setting, tests were carried for strict majority, $m = \frac{N}{2} + 2$, for the maximal group, $m = N$, and for majority–minority alternation $m = \frac{N}{2} + 1$. The strict minority, $m = \frac{N}{2} - 1$, rendered a singleton group for $N \leq 4$, so that experimental points could not be taken. The results for the mean values of the three observables are shown in figs. 4.14-4.16. Those plots show rather good scaling for these moderate sizes of the system, with all the observables exhibiting a linear behaviour and small slope.

Fig. 4.14 represents the time cost T_{join} as a function of the total number of nodes in the final group. T_{join} measures the time from the detection of a **Join** attempt until the installation of the following view, and thus includes the cost of forming, proposing and confirming a new view from the addition of a single node. Fig. 4.15 shows a related measurement: that of T_{comm}, the time it takes for the new node to become aware that it is included in the collective view. This quantity is necessarily larger than the former, since the time starts counting when the isolated node installs its *zero* view and stops after the node has received the confirmation of the collective view, and thus the measured period includes the time T_{join}, but also the cost of composing, sending and processing all the required information to update the history of the joining node. The extra time accounts for the cost of spreading the **Join** request and getting it accepted by the master of the formed group. Finally, fig. 4.16 shows the dependence of T'_{fail} with the size of the group. This represents the average cost of proposing and confirming a view due to the exclusion on a single member. In the case the excluded member was the master of the group, T_{fail} includes the cost of the master changing procedure.

In order to test the protocol in larger systems, multiple nodes had to be launched on each available machine. The physical cluster available for most of our tests consisted on four PCs (Intel® Pentium® 4, CPU 2.80 GHz), so that N was chosen as a multiplicity of 4, $N \in \{20,\ 40,\ 60,\ 80\}$. For each size, tests were run with $m = \frac{N}{2} - 1$, $m = \frac{N}{2} + 1$, $m = \frac{N}{2} + 2$ and $m = N$. The last two values correspond to strict majority and appear as a single series on the plots. The experimental results are contained in figs. 4.17-4.19.

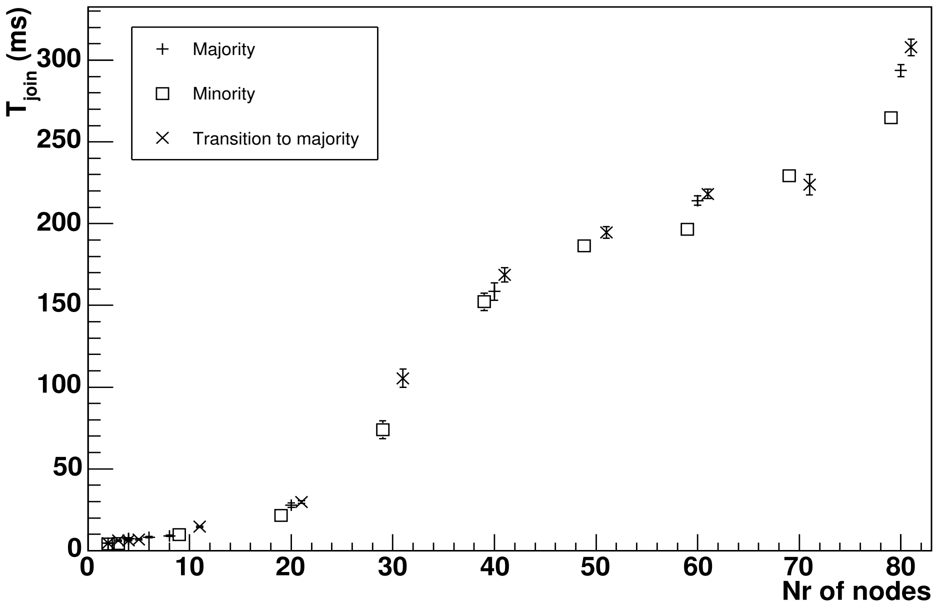


Figure 4.17: Mean value of T_{join} as obtained by execution on 4 PCs.

These plots show the increasing of the time cost of operations with the number of nodes that form the view. The scaling is faster for these larger system sizes as compared to the realistic settings described above, which are also shown in the graphs.

An important factor that must be taken into account when analysing this increment is the fact that the experiments were run on a test of only four physical nodes. Therefore, for system sizes of 10 nodes and beyond, several nodes were running on the same machine. The number of nodes launched per physical host does obviously affect the time cost of any operation. This was explicitly checked by using eight identical PCs to launch a system of $N = 8$ nodes and measure the three observables T_{comm}, T_{join} and T_{fail}, and then repeating the experiment using only 1, 2 and 4 physical hosts. The results are pictured in fig. 4.20, which shows a linear scaling of our observables with the number of nodes running in the same physical machine.

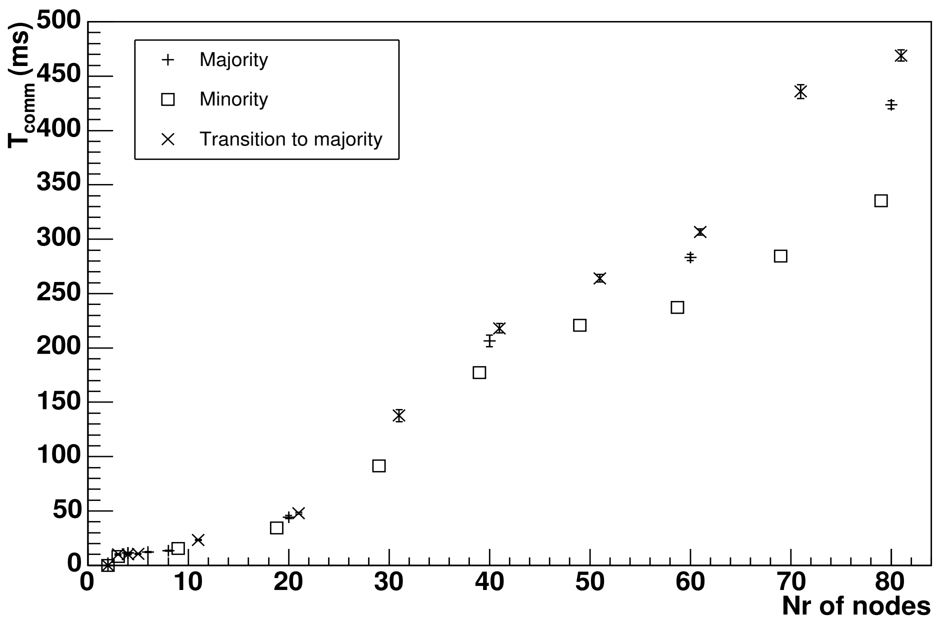


Figure 4.18: Mean value of T_{comm} as obtained by execution on 4 PCs.

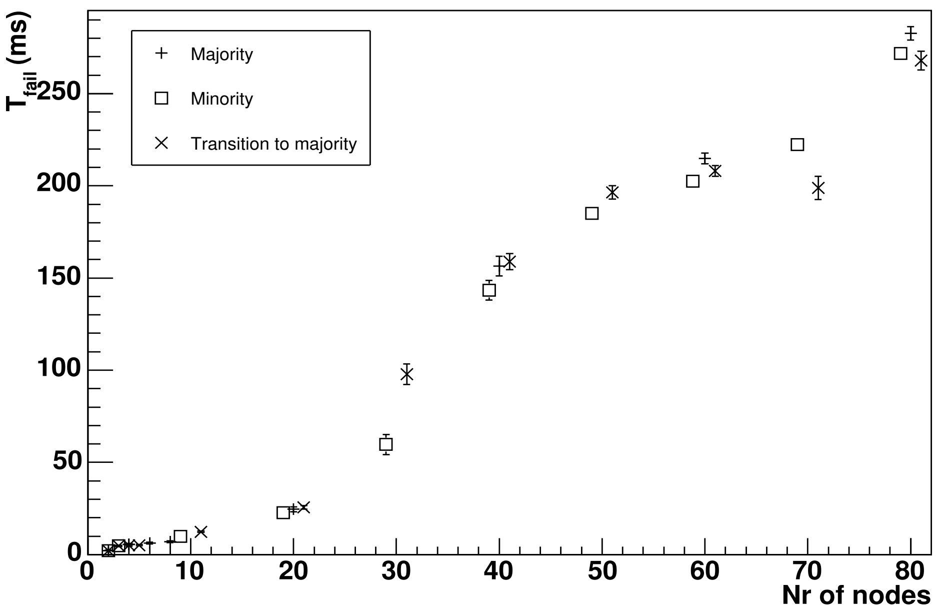


Figure 4.19: Mean value of T_{fail} as obtained by execution on 4 PCs.

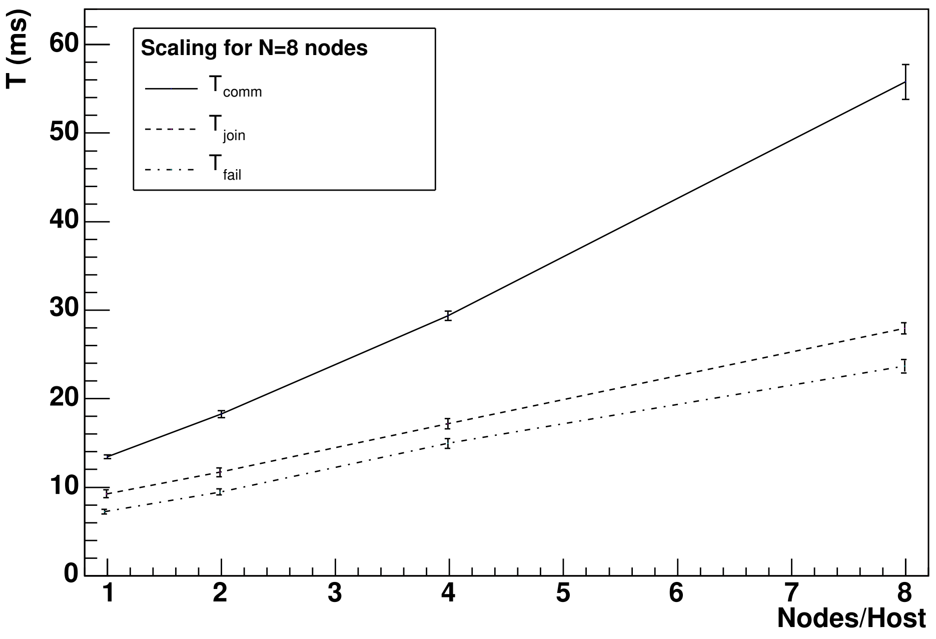


Figure 4.20: Scaling of time costs with the number of simulated nodes per physical host, for a cluster of $N = 8$ nodes.

4.5 Properties of HMS: Additional Uniform View Agreement

The HMS protocol fulfils the basic specification of the GMP in 3.1.2.

The major contributions of HMS, different to most existing specifications of membership services, are the following.

- Specific treatment of majority and minority operations, explicitly described in the specification of the protocol.
- Capability of losing and recovering the majority due to partition and remerging of the network, and to resume normal operation.

Besides the basic properties in 3.1.2, HMS provides an additional property related to the latter feature. This property can be enunciated as follows.

Property GM.6 (Uniform Agreement on Majority Views)**.** *A majority view is never installed in a node if there exists a previously committed majority view whose commitment is not known to this node.*

This means that the group majority history — the only one that is unique, being the group partitionable — is maintained by all the nodes in the system, as they take part of the majority. A single node may have an outdated information on the trajectory of the majority group while it is disconnected from such majority, but it will be brought up to date before installing a fresh majority view. Such guarantee is achieved thanks to the use of stable storage to keep the information on installed majority views, and to the subprotocols of partition merging and view history matching.

The usefulness of this property is clear when the group losses the majority due to failures or partitions, so that all the surviving nodes are left in minority operation. When the failed nodes or channels are repaired, and a new majority group is formed, the *Uniform Agreement* property ensures that each participant will be able to decide whether it missed some installed majority view, and to identify which members were present in the latest majority group without additional communication rounds. For an application that needs to take special actions upon recovery of failed nodes and which distinguishes the majority situation, this feature implies a great simplification in the logic of the recovery operation, as illustrated by the following example.

4.5.1 Example Case of Use

In this section we describe a case of use to illustrate the usefulness of the property GM.6 for a real application that makes use of the membership services.

A typical example might be given by a distributed database replication system. Let us consider a set of database servers in which the database is fully replicated, and picture an update–everywhere replication scheme [67] in which any node may receive and serve client requests. Thus a client may connect to any server to initiate a transaction, which will in general be composed of a series of read and write operations, and will be terminated by a *commit* or *abort* decision. Coordination is required among the replicas in order to ensure atomicity regarding the last decision: either all or none of the replicas must commit a given transaction. Moreover, synchronisation among servers is also required to guarantee serialisability of different transactions.

Replication protocols are easier to implement on top of group oriented services [68]. Detection of replica failures or virtual synchrony support from group communication services is helpful to provide the desired guarantees in case of failure. With access to membership information, the database application can react to the various failure patterns and recover to continue providing its service.

Let us consider then a particular replication protocol in which client requests may be received by any server. Let us also assume that this application runs on top of a partitionable membership service (which may be part of a group communication suit). In order to ensure consistency, progress is allowed as long as the group is in majority. Thus, when a node is partitioned from the group, it must stop serving client requests, so that only the majority group continues to operate and make changes to the database. When the node rejoins, it must be brought to the most up-to-date state before being able to resume normal operation, i.e. to receive and serve client requests normally, and the application must include the necessary recovery protocol to allow the restoration after failures.

Generally speaking, if we are dealing with a partitionable system, in which recovery after failures is supported, when a node rejoins the majority group after being partitioned or failed it must receive the latest updates applied by the majority group since its decoupling.

A special situation arises, however, when the network partitioning or the combination of failures destroys the majority, so that the surviving nodes are left in minority groups and the application blocks. After some of the failed nodes recover, or when the partition disappears, the majority can be recomposed. Then it is necessary to reconstruct the history preceding the majority loss, so that the most up-to-date server can be identified and instructed to update the others.

In principle, the replication protocol would receive a notice from the membership service about a majority view being installed. Every member of such group would then be aware that it is joining a majority group and that majority updates may be missing. However, none of the nodes has stayed in the majority group across the transition, neither knows whether any of the others took part in a later majority view than itself. Then at least one dedicated communication round is required in order to figure out which of them has the most up-to-date information and is responsible for updating the rest.

The property GM.6 of HMS avoids such a round, as each node will know, at the moment of installing the new majority view, all the previously confirmed majority groups. With this guarantee, every member will have access to the identities of all surviving participants of the latest majority group. This allows the updating phase to be initiated immediately, without need for additional communication.

Other more sophisticated examples might include role inheritance (either for replication management, or for other purposes) and role recovery after failures are repaired.

Chapter 5

Halo Membership Service

This chapter presents a different membership service for large scale systems, intended in particular for the very common scenario of client–server architectures. The system setting of our interest consists of a reduced, preconfigured group of nodes serving requests from a set of external clients whose identities and number are not predetermined, and which will typically connect over a WAN. From a practical point of view, this is the typical scenario that appears on the interaction between clusters devoted to offer highly available services and clients that connect to access to those services.

The problem involves thus the interaction between two node sets with very different properties. The lifetime of client–server connections is generally much shorter than that of server–server connections, which are expected to last for long periods of time in order to implement the required services. Moreover, the failure rate that affect clients — i.e. client node failures and failed communications between clients and servers — is expected to be at least one order of magnitude larger than the failure rate observed in the group of servers.

Here an extended specification of the membership problem is proposed for this kind of environment. It takes into account the different consistency properties that each type of node requires from membership information in such scenarios. We also present a practical service, HaloMS [62, 69, 70], that fulfils the specification and provides the required guarantees.

5.1 Large Scale Client–Server Scenario

The client–server scenario is a commonly occurring one in distributed systems, and in particular in the context of highly available services. Fault tolerant services are usually built over a reduced set of stable computers interconnected over high speed and highly reliable networks. Even though failures have to be addressed, they are expected to be infrequent. In contrast, clients that access those servers tend to do so through much less reliable and much slower networks (e.g. the Internet, wireless networks, etc.), running over much less stable hardware and system software. Besides the different speed and stability properties, the number of clients accessing services tends to be much bigger than the number of servers cooperating to provide them, and additionally, the duration of client–server interactions is usually very short compared to the duration of inter–server connections. From the point of view of simplifying the development of highly available applications or components, the deployment of a group membership service that provided strict semantics to all nodes would be the most favourable scenario. Nevertheless, such an attempt will necessarily face the difficulties inherent to a large scale, highly dynamic environment.

In spite of being a most useful tool for providing high availability, the usage of group oriented services in practice is usually constrained to small environments, formed by a reduced group of elements. Most of the existing classical group support systems described in section 2.2.1 are devoted to environments with low change rate, being the group members connected over a LAN, or at most over several interconnected LANs [36]). The reason for these limitation lies in the difficulties mentioned in 2.2.2 to develop scalable group membership and group communication protocols for the large scale or wide area networks.

Despite these difficulties, the availability of the group services (membership, group communication, consensus) is highly desirable also in large scale systems, as prove the different attempts that have tried to achieve scalable groups, described in section 2.2.2.

One of the first conceptual approach to large–scale group support, proposed by Babaoğlu and Schiper in [21], relied precisely on the differentiation of node roles, distinguishing servers, clients and sinks.

Regarding practical implementations, although various approaches exist to provide group support in the large–scale, as described in section 2.2.2, none of them would be suitable for client–server architectures. Just to mention a few, Spread [14] is aimed to provide strong guarantees regarding group changes and message delivery to all members in the system, and although addressed to WAN environments, it can only cope with a limited number of nodes; InterGroup [17] addresses the issue of scalability by distinguishing between active sender and receiver nodes, and membership agreement is run only among the former; Xpand [39] improves the performance over large groups by relaxing the consistency guarantees, but this is done as a function of the application requirements, and the same guarantees are provided to all system members.

Whereas from the point of view of the applications developer it would be desirable to have an homogeneous membership service providing accurate membership information to every single node, the associated cost is clearly excessive when dealing with big enough groups. On the other hand, client–server interactions that occur in a setup as the one we envision do not generally require precise information about membership. On the contrary, the consistency requirements of client nodes differ from those of servers [71]. As argued in [72], the model that best fits client–service interaction when the set of clients is large and dynamical is that of open groups, where clients are external to the group of servers. For some applications, it could more appropriate for the servers to maintain consistent views about the client set.

5.2 The System Model

Consistently with our general considerations in Chapter 3, the system is modelled as asynchronous and partitionable. It will be composed by a number of processes (or nodes) that communicate by means of message passing. Failures may happen to nodes (as crashes) or to network channels, and partitions may also occur. After crashing, a core process is allowed to recover and rejoin the group. However, for client nodes we consider the model of crash/no recovery, since client connections are typically short–lived. The system is equipped with a local failure detector module at every node.

As already mentioned, we focus on a particular scenario, in which two different sets of nodes interact. On one hand, there is a reduced preconfigured group of servers, which can be modelled as in section 4.1, and that will be referred to as *core*. On the other, there is a group of external clients that we call *halo*, whose identities and number are not predetermined, that request some service from the core. The connection of clients to the group of servers takes place over a WAN. All these observations are common when facing the design of fault tolerant systems, and can be considered as the common pattern that appears in cluster–like systems offering services through a WAN. In our model, we also assume that clients do not need to access other clients, so that the only relevant interactions are server–server activity and client–server accesses.

Although failures may happen to every node and connection in the system, the scenario under study allows us to assume that client–server connections are subject to more frequent failures than those internal to the group of servers.

As in section 4.1, the model includes an assumption regarding the dynamics of the *core* group, namely that within the server group there exist failure-free intervals of time long enough for the failure detector to stabilise and a full reconfiguration to take place. Such an assumption is not required for the group of clients, as their membership information is not required to satisfy the same strict liveness guarantees.

The large scale scenario we have depicted implies that a special effort is required to efficiently handle a large amount of unstable clients accessing a reduced set of servers. Efficiency is specially relevant when no failure affects the group of servers. When failures occur, servers have to reconfigure fast, even if a large amount of clients were connected to the faulty servers. Additionally, client connections, disconnections and failures also have to be treated efficiently within the group of servers, so that the remaining clients can be served with no significant loss of performance.

5.3 The Client Membership Service Specification

This work closely follows the idea of using different roles as proposed by [73], but applying them to group membership, independently of group communication. This is to be consistent with the modular architecture described in section 3.3.

The goal is to define a service responsible for managing the most typical information needed by clients in order to access servers, and the most frequent information that servers need about clients. Throughout the following sections we will assume that servers require strictly consistent membership information about themselves, and that is provided by a classical membership service fulfilling the basic specification.

The only information clients are assumed to know is the network address of at least one of the servers. Given this starting hint, clients will be provided with regular information about the server group composition. Such information will be delivered to each client as view change notifications in a similar way to the delivery of view change notifications within the server group. The only distinction will be liveness.

As a remarkable feature, in this approach the client group membership information is explicitly addressed, instead of being considered as part of the common shared state that has to be handled by upper software placed at the servers. In a certain sense, this work can be considered as a practical realisation of the ideas of maintaining distinct roles for servers and clients to support groups in the large scale in [73], while following the modular approach that separates group membership from group communication. With such a component based architecture, a group membership service for large environments should be available as a convenient building block to further develop either virtually synchronous group communication primitives or any other kind of recovery protocols.

Our approach relies on providing similar group membership interfaces to clients and servers. This interface, which was shown in figure 3.1, is general enough to allow the development of a variety of membership services, as already discussed, while being powerful enough to enable the development of upper components. The asymmetry of the problem regarding core and halo members implies that the concept of membership group notifications carries different meaning for servers and clients, and that those notifications are delivered with different semantics. The core group, i.e. that of servers, requires a group membership service satisfying the classical GMP, as enunciated in sect. 3.1.2. This can be provided by the HMS protocol described in the previous chapter or by any other classical membership service that fulfils such specification. In this chapter we are interested in the information related to client membership, which will not be provided as conventional views, but will require the definition of specific membership events.

Observations about client–server interactions and client–server software architectures, provide the means to relax the strict and classical membership semantics for client membership information, what constitutes the basis for the specification of a client membership service. Specifically, clients do not need to be aware about the presence of other clients, but rather some notion about the server group configuration that enables them to contact server nodes as needed. In contrast, server nodes need precise information about the rest of the servers, and accurate but possibly delayed information about clients. In this way, server applications with fault tolerant capabilities which are common for cluster setups will be enabled.

Fig. 5.1 illustrates these concepts by showing an example of the different knowledge each member holds about halo membership at a given moment. As depicted, while all core nodes share a consistent view of the core group, their local information about clients may vary. On the other hand, each client must only know about core nodes that already know about the client.

5.3.1 Notation

In the remaining of this chapter we will be referring to two different types of membership information. Regarding the membership of the group of servers, since it conforms to the classical specification of the GMP, we will use the traditional terminology in the context of membership services, already introduced in 2.1.3 and extensively used in the previous chapter.

The term *view* will thus refer to an image of the core group membership held by a particular core node. Since the group of servers is modelled as partitionable, an explicit distinction may be done between majority and minority views.

As discussed above, in the particular scenario we are concerned about two sets of nodes are defined: the core, or central group of servers, and the halo, i.e. the unbounded and dynamically changing set of client nodes. Each group requires different membership information regarding the other. In that sense, we introduce the concepts of *halo view*, to denote the group of clients a given server node knows about, and that of *membership horizon*, to denote the knowledge a client has about the group of servers.

5.3.2 Formal Specification

In order to specify the halo membership problem, we need to state which series of guarantees we require regarding the information delivered to and about the connected

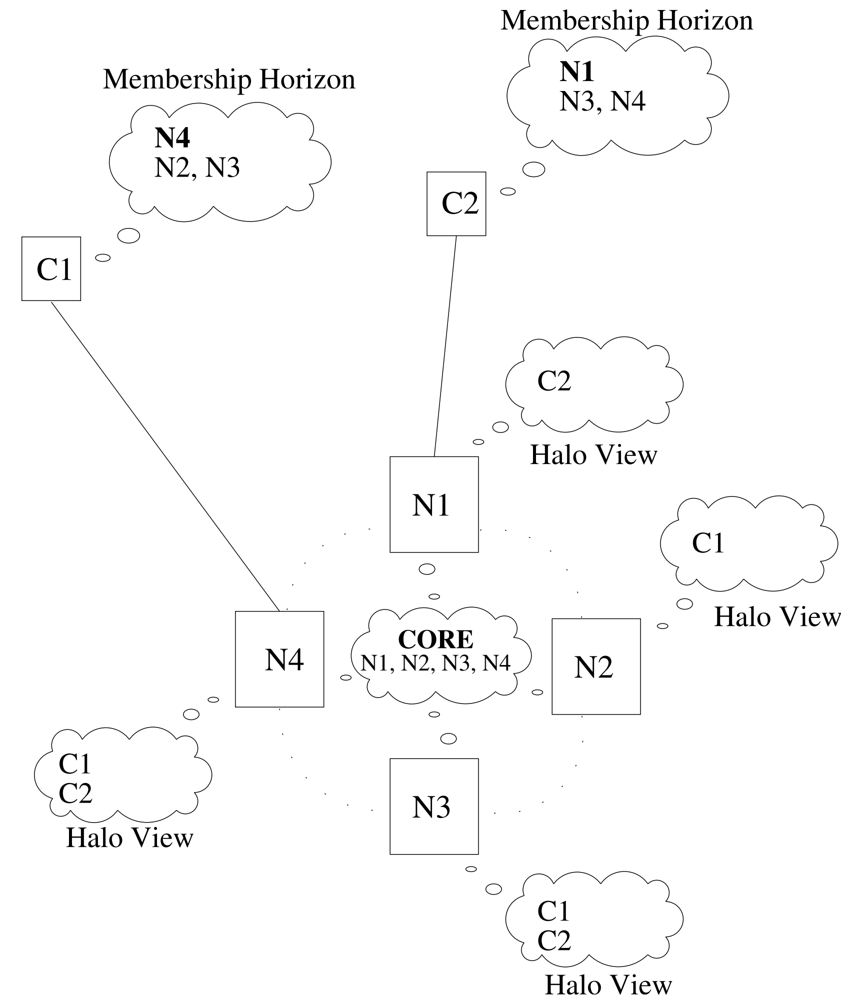


Figure 5.1: Contents of halo membership information maintained by server and client nodes at a given time.

clients. These requirements should be less restrictive than those of the core membership service, but enabling the membership service to serve as basis for the construction of upper highly available components.

The requirements can be expressed as the following list of properties, that acts as the problem specification.

Property HM.1 (Unicity). *A client receives a different unique identifier each time it joins the group.*

Whenever a new client connects, the core group must reach agreement on the joining, and this client must receive information on which core nodes are capable to be contacted for its requests. Thus, we want that a client receives a different unique identifier each time it joins the group, that can be used to identify it within the group for the time the connection lasts.

Property HM.2 (Validity). *A client node does not include a core member in its horizon unless the core node included the client in its halo view.*

A client node should not be able to communicate with a core member that has not yet known about this client's connection. Therefore, property HM.2 requires that a client's *horizon* includes only core members that already contain the client in their *halo views*.

Property HM.3 (Halo Consistency). *Two different core members do not disagree on the identifier of a particular client.*

This property prevents the client from receiving a double identifier in case it disconnects and tries to connect again before all core members have properly released the first identifier.

Property HM.4 (Halo Liveness). *If a client fails or disconnects, its identifier will eventually be invalidated.*

Client identifiers are assigned dynamically as clients connect to the group, and they are used to identify the client for the duration of its connection. Therefore, once the client disappears from the group, its identifier is no longer in use and should be removed from all *halo views* of core members. This is to guarantee that if the client fails or disconnects, the core group will eventually stop trying any communication with it. Any client related information may be safely removed from the system from this point on.

Property HM.5 (Horizon Liveness). **(i)** *If a core member included in a client's horizon fails, the client will eventually update its horizon to exclude the node.*

(ii) *Conversely, if a core member joins the group and both the client and the new member stay connected, the client will eventually include the core node in its horizon.*

Finally, we specify some degree of liveness in halo membership information, enough to ensure the progress of the system after changes occur to the core group that agreed on the joining of a certain client. The first liveness condition to be satisfied ensures that clients will not indefinitely try to contact a failed core member. This does not suffice to guarantee that in case that a core node disconnects from the group it does not keep communicating with its known clients. Thus in case of partition in the core group, two different partitions may know about the client's identifier and the client may receive messages coming from both. The membership protocol running on core nodes cannot be aware of this circumstance in neither of the groups. Hence, the client side of the protocol will be responsible to discriminate between such messages and remain joined to one of them while discarding the other. The decision will depend on the policy about partitions of the particular protocol that implements this specification.

Conversely, the second part of property HM.5 ensures that new core members get to be known by connected clients, to maintain a client's connection after a number of changes happen to the core group. After a client has been granted an identifier, all core nodes knowing about it might fail, so that the client would exclude them from its horizon and this could be reduced to an empty set. The property HM.5 guarantees that if new members were added to the core group, they will be updated with respect to the client's connection and the latter will not be lost.

5.3.3 Scenario of Applicability

To illustrate the usability of such an approach, let us picture a particular scenario of client–server interactions, as shown in fig. 5.2. There a certain client, $C1$, is issuing a number of invocations to the group of servers. Such invocations do not need to be served by a single core member, but different servers can be involved in interactions with $C1$. Moreover, during the process of serving each invocation, a core node could launch one or more invocations to other servers, as depicted in the figure.

In case of failure of the client, there may be a number of pending invocations that servers should manage. In a scenario as the one shown, the tasks needed to monitor the client's connection and react in case it is lost have to be repeated by all involved servers. With a client membership service, instead, the client connection is managed by the group at the membership level, therefore allowing a unified treatment of the fault and simplifying the logic of the application.

Moreover, if the service has to tolerate server failures, the information on the client's connection will have to be replicated by the servers. In case of failure of the server

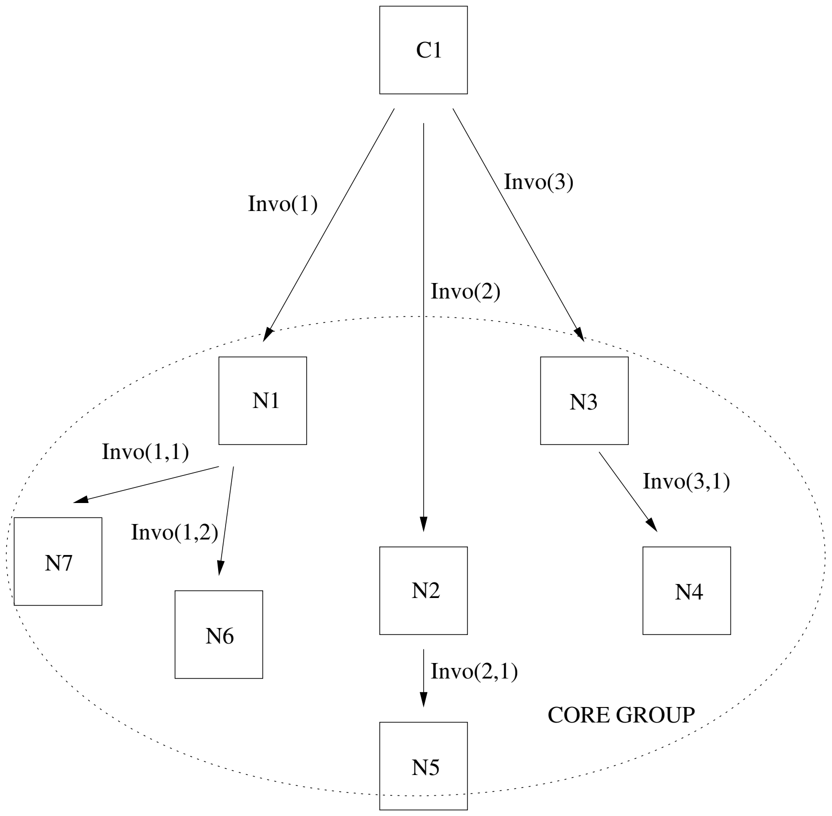


Figure 5.2: Example scenario for a client membership service.

that received an invocation, this allows other servers to either continue attending it or at least recognise redundant invocations. The replication of the client's connection can also be handled at a lower level by the client membership service, what makes it more transparent for the application.

5.4 HaloMS

This section describes HaloMS (*Halo Membership Service*), a particular membership service designed to fulfil the above specification of a client membership service. The HaloMS is devised as a component to be laid on top of an existing partitionable core membership service that provides the basic membership guarantees specified in 3.1.2.

5.4.1 Architecture

The specification of the client membership service is independent of the particular classical membership service that maintains the information on the core. Thus, the HaloMS service could be deployed on top of different classical membership services. But in any case, it will make use of the information provided by them in the form of views.

According to the general approach of a modular architecture, with well–defined interfaces to ease the interaction among components, the HaloMS should offer the generic interface of an `IMembershipMonitor` (see fig. 3.1) to upper components, and the interface of `IMembershipListener` (see fig. 3.2) to the underlying classical membership service.

Contrary to a core membership service, the HaloMS involves interaction among servers and clients, and must be running on both types of nodes. Servers will be informed by the HaloMS about clients joining or leaving the group, and conversely clients will be notified about changes to the group of correct servers that know about them. This means the HaloMS will have modules running on each type of node. From the point of view of the design, both types will offer the same interface of `IMembershipMonitor`. The decision of unifying server and client group changes under a similar group membership interface allows an easier development of upper software components that need to react to group changes.

Having the same interface, however, servers and clients require different implementations of the group membership service. Core nodes, as shown in figure 5.3, will include a traditional membership service (*Core Membership Service*), and a specific HaloMS component to maintain client group membership. The Halo Membership Service will use the information provided by the Core Membership Service — conforming to the

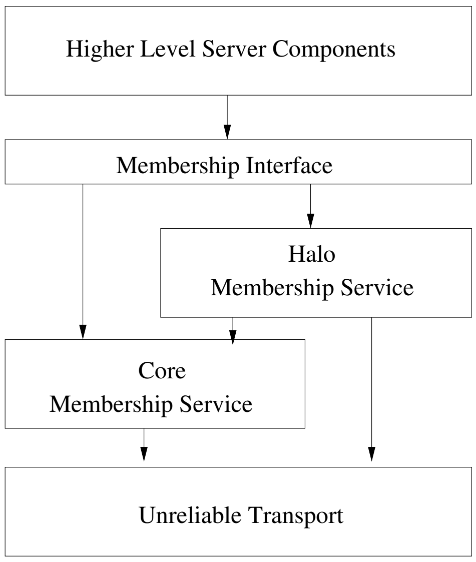


Figure 5.3: Architecture of membership components in a core node.

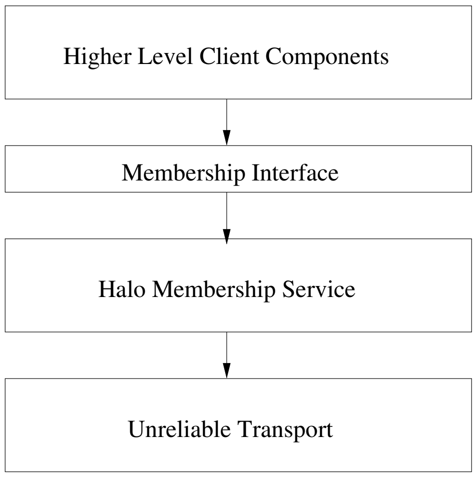


Figure 5.4: Architecture of membership components in a client node.

general interface of `IMembershipMonitor` — and will register as `IMembershipListener` to be notified about *committed* core views. Correspondingly, it will inform higher level components about changes to the *halo view* by defining specific membership events. As shown in the figure, in the most general case higher level applications will have access to both types (core and halo) membership information through the same interface, but exchanging specific event types for each of them.

On the other hand, on client nodes no classical membership service is required. The only membership information will come from the corresponding module of HaloMS. As shown in the figure 5.4, such information will be equally provided through the generic interface.

5.4.2 Basic Elements

The HaloMS protocol is based on the circulating token algorithm for mutual exclusion [74]. In its basic version, the token is generated by a majority core group and circulates along a logical ring formed by the view members. In this section we describe in detail the operation of this membership service.

Node Identifiers

Within the core set (see 4.1), every node has a predefined identifier, known in advance by the whole group. On the contrary, the identities of halo nodes are not predetermined. Therefore the halo identifiers cannot be assigned a priori.

A unique integer identifier is thus assigned dynamically to each halo node as it joins the group. Thereafter the client node is referred to by such identifier for as long as it is connected to the group. Moreover, no client request may be served until the identifier has been granted. After the node leaves the group, its identifier has to be invalidated. In any case, the halo identifiers have to satisfy the condition not to collide with core preassigned identifiers. Since the latter are known beforehand, HaloMS chooses client identifiers from the set of integers larger than the highest core identifier. The first client identifier is chosen as an integer far enough from the reserved core set, and the following integers are subsequently used.

The continuous increasing of the identifier is not a problem in a practical situation,[1] but it is also possible to recycle halo identifiers after they are invalidated, if the rank of identifiers is limited for saving space.

[1]Notice that a four–bytes identifier would allow a number of connections as large as $\approx \text{Ø}(2^{32})$ nodes.

State

The HaloMS protocol, as already discussed, runs different modules on servers and clients, and therefore different state is maintained on each type of node.

Core nodes

- Each core node maintains partial information on the connected clients, in the form of a *halo view*. This consists of a set of entries, each of them corresponding to a known client. A given entry in the *halo view* has the form

  $$[Id,\ address,\ N_{\mathrm{r}},\ status,\ V,\ V_{\mathrm{u}}]\,,$$

  where Id is the assigned client identifier, *address* holds the halo node's physical address, and N_{r} the identity of a core node that acts as its *representative*. The field *status* indicates the level of confirmation of the client's connection to the group,

  $$status \in \{proposed, committed, definitive, removed\}\,.$$

  On the other hand V is a core view identifier whose meaning with respect to the client connection may be different depending on the particular value of *status*. As will be described in detail later, after a client status is set to *committed*, its representative may decide to perform an updating round to refresh the core group information on this entry (as required by property HM.4 of the formal specification). The field V_{u} will be not empty only when an updating round is running, and in that case it will contain a view identifier related to the updating procedure.

  When a core node starts up, its *halo view* is empty, and successive token passes update it as appropriate. Since the HaloMS protocol allows integration of clients only in a majority core group, the *halo view* is reset every time the node abandons the majority.

- Moreover, each core node running the halo component keeps a copy of the last token it received. This is needed for recovery purposes.

Halo nodes

The HaloMS needs also to maintain specific state in the client nodes. This is, however, much more limited than the state in core members. The main element in a client's state is its *membership horizon*. This holds a set of core members that are aware of the client's presence in the group and agree on its identifier. Besides, the client keeps also its assigned identifier, to use it in its communications with the group.

When a client starts up, it is only assumed to know the network address of at least one of the servers, which will be used to try to establish a connection to the group. The client's *membership horizon* can be initialised only after the client's connection attempt has been answered by the core group. Such answer provides the initial *membership horizon*, a snapshot of the core membership when agreement was achieved on the identifier granted to this node. Further updates to the *horizon* do not necessarily reflect all core group changes, and thus it will in general not coincide with an installed core view.

Messages

As the rest of protocols described in this work, the operation of HaloMS is also regulated by specific messages. The client may send the following types of messages.

- **JoinReq(**$address$**,**ref**)** is sent by the client in order to establish a connection with the core group. The message contains as first argument the physical address of the client, to allow its identification by core members. This message is sent in two occasions to establish the connection. First the client addresses a **JoinReq** with empty ref field to some core node known in advance as the first contact point. Before admitting the client, the contacted core node will try to assess the freshness of the joining request providing an integer reference that will be included by the client as the ref argument of a second **JoinReq** message.

- **LeaveReq(**Id**)** is sent by a client in order to announce its voluntary leave. It contains as the only argument the assigned identifier Id by which it is recognised within the core group.

Besides the client's messages to ask for and dismiss a connection, the core nodes send the following types of messages.

- **JoinQst(**ref**)** is sent by a core node in reply to an initial **JoinReq** from a certain client. It includes an integer reference ref that is used to verify the currentness of the client's request.

- **Horizon(**Id**,** *list***)** is used to notify a client about the core members it may contact, when it finally has been accepted by the core group. It contains the assigned identifier, Id, and a list of nodes, *list* containing the composition of the core group that agreed on it. The same type of message is used to notify to the client an update on the set of core members it is allowed to contact, as this set gets modified by failures or new incorporations within the core group. In such case, the *list* will not be equivalent to a core view, but only to a subset of the currently installed view.

- **Token(**Tid**,** Cid**, . . .)** circulates along the ring of core members that form a stable view. It holds an identifier of its own, Tid, which consists of an integer counter, and the node identifier of the core member that generated the token. The counter is increased every round and whenever the token is regenerated. It also carries the first available identifier for clients, Cid, which is increased as members of the ring use it for new clients. Then, a series of entries follows, each of them indicating an action to be taken on a particular client. Any member may introduce an entry in the token, thus acting as initiator of the round corresponding to that particular entry. Entries thus carry the identifier of the initiator of the round, N_{ini}, the client's assigned identifier Id and optionally some specific information for that action, as listed below.
  - **ProposeTE**(N_{ini}, Id, $address$, N_{rep}) is used to ask the core to admit a new client, proposing an identifier for it, and carries the physical address of the client and its *representative* identifier, N_{rep}.
  - **ConfirmTE**(N_{ini}, Id) is used to notify the definitiveness of the addition.
  - **UpdateTE**(N_{ini}, Id, $address$, N_{rep}, V_{from},V_{to}) is basically used to let newly joined core nodes about already connected clients. It carries the same information as **ProposeTE** entries plus the identifier of the last view that committed the client, V_{from}, and the current core view identifier, V_{to}, indicating the view to which the client is being updated.[2]
  - **RemoveTE**(N_{ini}, Id) is used to announce the leave or failure of a connected client to the core.
  - **DefinitiveTE**(N_{ini}, Id) is used to notify the end of a confirmation or removal round. In the second case, it instructs the core to release the identifier of a left client.

  The token is first generated by the Master of the core group when the majority is attained, and has to be regenerated by a specific recovery subprotocol each time the majority core group goes under a reconfiguration.

- **RToken(**N**,** $lastToken$**)** is a *recovery token* used after a reconfiguration of the core group in order to regenerate the token with the most up-to-date information. It circulates along the remains of the former majority view in the opposed direction to the regular token, and contains as arguments the identifier of the sending member and the last regular token received by that node.

Membership Events

The HaloMS service must offer also the interface of generic membership services. In particular, to higher level components in server nodes it offers the interface of

[2]This will be needed for regenerating the token after a view change.

`IMembershipNotifier`. Notice that the `IMembershipMonitor` interface, with `join` and `leave` methods as shown in fig. 5.3 is jointly offered by this and the underlying core membership service, so that HaloMS does only need to implement the notification of its own membership information. This is done by means of a specific event in the algorithm which in the generic interface corresponds also to a definite type of `MembershipEvent`.

- `halochange` event is produced by the HaloMS protocol at a core node when it determines the confirmation or removal of a client node. In the generic interface, the type MBSHIP_HALO is specifically defined to this end. Such an event may carry as argument the current composition of the *halo view* or (for better scalability with the number of clients) only the added and removed clients from the last notification.

In client nodes, on the contrary, the HaloMS service is in principle responsible of all management of membership information. Thus it offers to client–side applications the whole interface of `IMemberhsipMonitor`. It also defines a specific event to communicate membership changes to its listeners.

- `horizonevt` event is produced by the protocol at client nodes whenever the node gains knowledge about some change to its *membership horizon*. In the generic scheme of common interfaces, this is notified to client `IMembershipListeners` by means of the specific type MBSHIP_HORIZON, which should carry as argument the currently known composition of this client's *horizon*.

Besides its own notifications, the HaloMS service interacts, via the standard interfaces, with the core membership service. Thus it reacts to the membership events defined in the previous chapter.

5.4.3 The HaloMS Protocol

As for the HMS protocol described in the previous chapter, the formal specification of the HaloMS protocol was made in the formalism of I/O Automata. It is included as an appendix (B.1) to this work. This section presents, nevertheless, the specification in terms of states and transitions.

Core Members

The part of the HaloMS protocol run by core members can be specified as the five states shown in fig. 5.5, and transitions among them are caused by membership

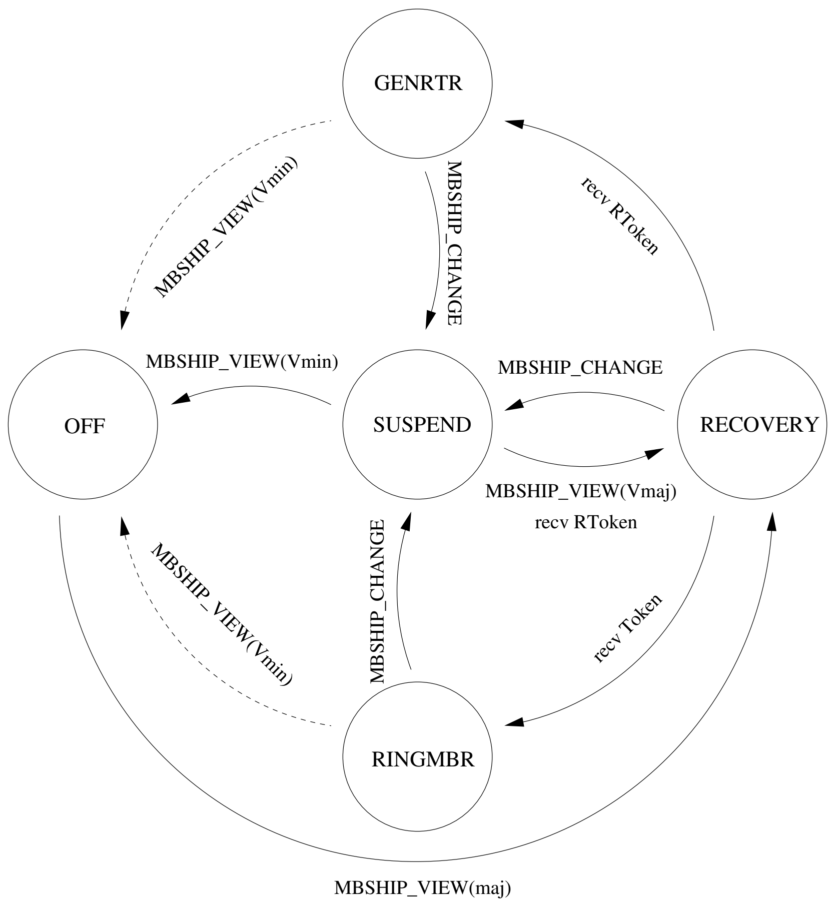


Figure 5.5: States and transitions of the HaloMS protocol at core members. The dashed lines represent transitions that would only take place in case the core membership service did only notify view installation events.

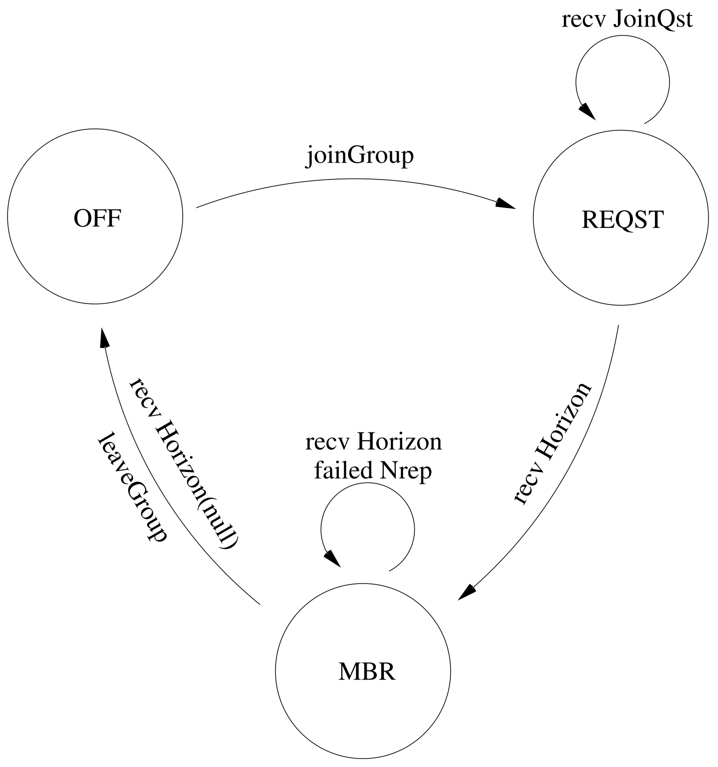


Figure 5.6: States and transitions of the HaloMS protocol at client members.

events notified by the underlying core membership service and by the specific HaloMS messages described above. These states are the following.

- OFF: In this state no *halo view* is maintained, and the Token is not circulating along the members of the view. This happens when the node is a member of a minority view. In particular, it is also the initial state.
- RECOVERY: It is the state adopted after a majority view has been installed, in which the token may be set to circulate again. This state performs the necessary operations in order to identify the most up-to-date member and to reconstruct the token.
- RINGMBR: In this state, a node runs the basic algorithm to add or remove clients, as the token pass allows it to do so.
- GENRTR: This is the state of the node that takes charge of generating the token and setting it to circulate after a reconfiguration of the core group. Besides increasing the token identifier after each round, the basic operations are equivalent to those run in RINGMBR.
- SUSPEND: If the core membership service, as HMS, provides notifications about ongoing changes or tentative views, such a notice about the instability of the current core view may be used to stop the token circulation until a view is definitively installed and the ode can enter the RECOVERY or OFF state depending on the majority or minority character of the installed core view. This state, however, may be skipped if the underlying membership service only provides notifications about installed views. In such case, the transition from GENRTR and RINGMBR states would be directly to RECOVERY or OFF (see fig. 5.5).

Figures 5.7-5.11 show the pseudo-code for the various states, while a detailed but informal description of the different algorithms is present in the following sections.

```
algorithm HaloMS
type
  state_t = {OFF, RECOVERY, GENRTR,
      RINGMBR, SUSPEND}
  entry_st_t = {PROPOSED, COMMITTED,
     DEFINITIVE, REMOVED}
  halo_entry_t = (client_id, address,
     entry_st_t, rep_id, view, viewU)
  token_entry_t = {ProposeTE,
     ConfirmTE, DefinitiveTE,
     UpdateTE, RemoveTE}
  token_t = (Tid, Cid, list of
     token_entry_t)
  ring_t = (generator,sorted list
     of nodes)
var
  state :  state_t
  lastV : view_t
  view :  view_t
  haloV : list of halo_entry_t
  lastT : token_t
  recoT : token_t
  ring :  ring_t
  toProcess, toRegen :  boolean
begin
  view := null;
  lastV := null;
  haloV := empty;
  lastT := recoT := null;
  state := OFF;
  toProcess:=toregen:=false;
  while true do
     case state of
     OFF:      off;
     RECOVERY: recovery;
     GENRTR:   genrtr;
     RINGMBR:  ringmbr;
     SUSPEND:  suspend;
     esac
end
```

Figure 5.7: Basic algorithm of HaloMS.

```
algorithm off
begin
  lastT := recoT := null;
  ring := null;
  haloV := null;
  wait for event
  case event of
  MBSHIP_VIEW(V):
    lastV := view;
     view := V;
     if V is majority then
       state := RECOVERY;
     fi;
   esac
end
```

```
algorithm suspend
begin
  recoT := null;
  wait for event
  case event of
  MBSHIP_VIEW(V):
    lastV := view;
    view := V;
    if V is majority then
       state := RECOVERY;
     else
       state := OFF;
     fi;
   recv(RToken()):
     recoT := msg;
     state := RECOVERY;
   esac
end
```

Figure 5.8: Algorithms for the OFF and SUSPEND states.

```
algorithm recovery
type
var
  pred :  node
init
  pred := null
begin
  inherit representatives;
  if recoT!=null then
    if lastT=null then
      if recoT.token=null then
        state := GENRTR;
        ring := (thisNode,
          view.members);
        toRegen := true;
      fi;
    else if lastT.Tid>recoT.Tid then
      state := GENRTR;
      ring := (thisNode,
          view.members);
      toRegen := true;
    fi;
  fi;
  if state= RECOVERY then
    if lastT!=null then
      pred := immediate predecessor from
          ring ∩ view.members;
      send RToken(thisNode,lastT)
          to pred;
    else if thisNode is the lowest from
      view then
      pred := immediate predecessor
          from ring.members;
      send RToken(thisNode,null);
          to pred;
    fi;
    wait for event
    case event of
    recv(RToken()):
      recoT := msg;
    recv(Token()):
      lastT := msg;
      ring := (lastT.generator,
          view.members);
      toProcess := true;
      state := RINGMBR;
    MBSHIP_CHANGE:
      state := SUSPEND;
    esac
  fi;
end
```

Figure 5.9: Algorithm of the RECOVERY state of HaloMS.

```
algorithm ringmbr
type
  halo_changes_t = (client, action)
var
  haloChg :  list of halo_changes_t
  nextT : token_t
  nextCid :  integer identifier
  joining :  list of (address,ref)
  toAdd :  list of addresses
  toRemove :  list of client identifiers
  toUpdate :  list of client identifiers
init
  haloChg := null
  nextT := null
  joining := null
  toAdd := null
  toRemove := null
  toUpdate := null
begin
  recoT := null;
  nextCid := lastT.Cid;
  toUpdate built from heuristic algorithm run on haloV;
  haloChg := null;
  if toProcess then
    nextT, haloChg <- haloV.process(lastT);
    for Cli in toAdd
      haloV.add(nextCid, Cli.address, PROPOSED, thisNode, view, null);
      nextT.add(ProposeTE(nextCid, thisNode, Cli.address, thisNode));
      nextCid := nextCid+1;
    done;
    for Cj in toRemove
      nextT.add(RemoveTE(Cj,thisNode));
      haloChg.add(Cj,remove);
    done;
    for Ck in toUpdate
      nextT.add(UpdateTE(Ck, thisNode, Ck.address, Ck.rep, Ck.view, view));
    done;
    toAdd := toRemove := toUpdate := empty;
    nextT.setCid(nextCid);
    send Token(nextT) to ring.nextTo(thisNode);
    nextT := null;
    nextCid := null;
    halochange(haloChg);
    haloChg := null;
    toProcess := false;
  fi;
  wait for event
  case event of
  recv(Token()):
    lastT := msg;
    toProcess := true;
  MBSHIP_CHANGE:
    state := SUSPEND;
  failed(Cli):
    if haloV.getEntry(Cli) has rep=thisNode then
      toRemove.add(Cli.id);
    fi;
  recv(JoinReq(C,ref)):
    if ref=null then
      ref := new reference;
      joining.add(C.address, ref, timer);
      send JoinQst(tisNode, ref) to C;
    else if joining contains (C, ref) then
      toAdd.add(C);
      joining.remove(C);
    fi;
  joining.mbr timeout:
    joining.remove(mbr);
  esac
end
```

Figure 5.10: Algorithm of the RINGMBR state.

```
algorithm genrtr
type
  halo_changes_t = (client, action)
var
  haloChg : list of halo_changes_t
  nextT : token_t
  nextCid : integer identifier
  joining : list of (address,ref)
  toAdd : list of addresses
  toRemove : list of client identifiers
  toUpdate : list of client identifiers
begin
  if toRegen then
    nextT <- recovery(haloV, recoT);
    nextT.setTid(lastT.Tid+1);
    nextCid:=1+max(lastT.Cid, C proposed in haloV);
    for Cli in toAdd
      haloV.add(nextCid, Cli.address, PROPOSED, thisNode, view, null);
      nextT.add(ProposeTE(nextCid, thisNode, Cli.address, thisNode));
      nextCid := nextCid+1;
    done;
    for Cj in toRemove
      nextT.add(RemoveTE(Cj, thisNode));
      haloChg.add(Cj,remove);
    done;
    for Ck in toUpdate
      nextT.add(UpdateTE(Ck, thisNode, Ck.address, Ck.rep, Ck.view, view));
    done;
    toAdd := toRemove := toUpdate := empty;
    nextT.setCid(nextCid);
    send Token(nextT) to ring.nextTo(thisNode);
    nextT := null;
    nextCid := null;
    haochange(haloChg);
    haloChg := null;
    toProcess := toRegen := false;
  fi;
  recoT := null;
  if toProcess then
    proceed as RINGMBR, but before sending nextT, increase nextT.Tid;
  fi;
  wait for event
  case event of
  recv(Token()):
  MBSHIP_CHANGE:
  failed(Cli):
  recv(JoinReq(C,ref)):
  joining.mbr timeout:
    Proceed as in RINGMBR;
  esac
end
```

Figure 5.11: Algorithm of the GENRTR state.

Clients

The part of the HaloMS protocol run by clients is much simpler, as they do not need to coordinate among themselves, but every client does simply interact with the core group requesting its admission and receiving information about core members. In our approach, clients use a reliable point-to-point communication channel to connect to one of their already known set of servers. Whenever a client detects that its server endpoint fails, it will try to connect through a different server.

Figure 5.6 shows the corresponding states and transitions which specify the client part of the protocol. In particular, there are three different client states.

- OFF: This is the initial state, before anyone invokes the method `joinGroup` of the HaloMS. In this state the client does not have any contact with the core group.
- REQST: After `joinGroup` has been invoked, the client starts the process to enter the *halo view* of core members by requesting an identifier. It cannot be considered part of the group until it has received one from the core representative.
- MBR: It is the state of an accepted client node, which has already received an identifier assigned to it by the core group. The client says in this state until some higher application invokes the method `leaveGroup` and forces it to exit the group, or the representative core node notifies an empty **Horizon**, meaning that the client has been excluded, typically because of a loss of majority by the core group.

The informal description of client operation can be found in the following sections, while figs. 5.12-5.14 contain the pseudocode for this part of the HaloMS protocol.

```
algorithm HaloMS(Cli)
type
  state_t={OFF, REQST, MBR}
  link_t=(core node, link quality)
var
  state: state_t
  localId: integer identifier
  horizon: list of link_t
  startup : list of initial core
      contact(s)
  idx : integer
  endpoint : (node, ref)
begin
  startup initialised from
    configuration;
  state := OFF;
  endpoint := (null, null);
  idx := 0;
  localId := null;
  horizon := null;
  while true do
  case state of
    OFF: off;
    REQST: reqst;
    MBR: mbr;
  esac
end
```

Figure 5.12: Basic algorithm of the client part of HaloMS.

```
algorithm off
begin
  rep := null;
  localId := null;
  horizon := null;
  idx := 0;
  horizonevt(null);
  wait for event
  case event of
   joinGroup:
     endpoint := (startup.get(idx), null);
     send JoinReq(thisNode) to endpoint.node;
     state := REQST;
   esac
end
```

```
algorithm mbr
begin
  wait for event
  case event of
  recv Horizon(null):
    state := off;
  recv Horizon(rep,Id,mbrs):
    endpoint.node := rep;
    horizon.update(mbrs);
     horizonevt(horizon);
   failed(node):
     decrease node.quality in horizon;
   leaveGroup:
     state := off;
   esac
end
```

Figure 5.13: Algorithm of the OFF and REQST states of the client side of HaloMS.

```
algorithm reqst
var
  tref :  timer
begin
  localId := null;
  horizon := null;
  start tref;
  wait for event
  case event of
  recv JoinQst(node,ref):
    if node=endpoint.node then
      endpoint := (node, ref);
      send JoinReq(thisNode, ref) to node;
    fi;
  recv Horizon(node,Id,mbrs):
    localId := Id;
    endpoint := (node, mbrs);
    horizon := mbrs;
      horizonevt(horizon);
    state := MBR;
  failed(core):
    if core=endpoint.node then
      idx := idx+1;
      endpoint := (startup.get(idx), null);
      send JoinReq(thisNode, null) to endpoint.node;
    fi;
  tref timeout:
    idx := idx+1;
    endpoint := (startup.get(idx), null);
    send JoinReq(thisNode, null) to endpoint.node;
  esac
end
```

Figure 5.14: Algorithm of the REQST state of the client sede of HaloMs.

5.4.4 Basic Operation

In core nodes, the Halo Membership Service lies on top of the corresponding core component and makes use of the guarantees this provides. In a system as the one under consideration, the core group is formed by a reduced number of preconfigured nodes. The basic HaloMS protocol described here is designed so that client additions are allowed only when the underlying core membership service has delivered a majority view. As already mentioned, an agile treatment of client failures, connections and disconnections during regular operation of the core group is essential, since changes to the group of servers will be much less frequent. This subsection thus deals with the regular operation of HaloMS when no changes occur to the core, but it is in a stable view. The reaction of HaloMS to core failures and the recovery subsequent protocol will be described in detail in later sections.

The algorithm progresses as the token circulates along the logical ring of core members. The ring is composed by all confirmed members of the view, starting from the node that set the token to circulation. The generator role is played by a deterministically elected node (e.g. the master of the view), when the protocol is first started, and by the node that regenerates the token, after a reconfiguration. Within the ring, nodes are ordered according to their fixed node identifiers. In principle, other criteria can be used to compose the logical ring with the view members, but we choose this one for simplicity. When a node receives the token, it processes its contents, modifies them if appropriate, and passes it on, keeping a copy of the last received token, to allow recovery in case of reconfigurations.

Client Addition

A client trying to join the group will get in touch with one or several core nodes about which it should know in advance by configuration. To establish this contact it sends a **JoinReq** message to one such core node, including its physical address and waits for a response. If, after a certain time, the answer is missing,the client may retry to contact the same core member or another one, if known.

A core member receiving such request will construct an integer reference, ref, and interrogate the client about the validity of the request with a **JoinQst** message that includes ref as argument. While waiting for the answer, the core node locally saves the client's physical identity and the sent reference for later checking. When the client receives the core question, and provided it has not been already accepted by other member, it replies with a second **JoinReq** that includes the received reference, ref. This message, when received by the core member that was trying to validate the client's intentions, signals the start of the addition protocol within the core group, as up to this moment, all interactions have taken place in a one-to-one basis between the client and a core node.

When a core node that has validated the client **JoinReq** receives the token, it will act as the client's representative within the core group and propose its joining with the first available client identifier, carried by the token.[3] This node, say N_{rep}, will also be in charge of sending **Horizon** messages to the client once the identifier has been accepted by the group or whenever the set of core nodes known to the client has suffered some change.

When N_{rep} gets the token it processes it to check whether some other node had already proposed the addition of this client. Then, if that is not the case, and the client is not yet present in its *halo view*, N_{rep} will take the following actions.

1. First N_{rep} decides the identifier that is to be assigned to the client, by retrieving the first available client identifier from the token, so that $Id = Cid$. The available identifier that will be included in the token when it is passed on is increased, $Cid = Cid + 1$.

2. A new local entry is created and inserted into the *halo view* with contents

$$[Id,\ address,\ N_{\text{rep}},\ proposed,\ V],$$

   where V is the current view identifier.

3. An entry **ProposeTE**(N_{rep},Id,$address$, N_{rep}) is added to the token, where the last N_{rep} stands for the initiator of the proposal round.

4. Finally, the modified token is passed on along the ring.

Subsequent core nodes will add the same **ProposeTE** local entry to their *halo views* as they process the token. Then, when N_{rep} gets the token back, it will set the local *status* of Id in its own *halo view* to *committed*, it will answer to the client with a **HorizonTE**(Id, $V.members$) message containing the list of core members in V, and substitute the **ProposeTE** entry in the token by **ConfirmTE**(N_{rep}, Id).

During the next token round, the local *status* at the rest of core nodes is also changed to *committed*. When the round is completed, N_{rep} will set this *status* to $definitive$ and promote the token entry to **DefinitiveTE**(N_{rep}, Id). Finally, after the third round, the client will be labelled as $definitive$ at all core members, and N_{rep} will delete the entry from the token.

The addition of a client to the *halo views* must be performed in three steps, corresponding to the different token rounds just described, to enable the recovery and update of information in case the protocol is stopped due to a reconfiguration of the core group, as will be described later on.

[3]Depending on the politics used by the client to retry the **JoinReq**, several core nodes could be in the situation to start proposing the client addition. In that case, only the first one to get the token will act as representative whereas the others will dismiss their pending **JoinReq** when they check the physical address of the client is already being proposed.

Client Disconnection

The procedure to remove a client is similar but it only needs two token rounds. When the client voluntarily disconnects from the group, or fails, its representative N_{rep} appends an entry **RemoveTE**(N_{rep}, Id) to the token. Nodes which knew about the client's Id will set its *status* to *removed* in their respective *halo views* when the token passes them by. After the first round is completed, i.e. when N_{rep} gets the token back, the token entry is substituted by **DefinitiveTE**(N_{rep}, Id), and each node removes the client from its *halo view.* After the end of the second round, N_{rep} will remove the entry from the token, and the identifier Id definitely disappears from the group.

After the client is connected to the group, changes may happen to the core. In order to guarantee the liveness property HM.5, the representative will eventually decide to refresh its client's *membership horizon* so that it includes all members of the current view, say V. To this end, and assuming the client's entry has reached the *committed* status in a view V_{old}, N_{rep} will add an entry **UpdateTE**(N_{rep}, Id, *address*, N_{rep}, V_{old}, V) to the token. When the remaining core nodes receive the token, those that did not know about this client will add an entry

$$[Id,\ \textit{address},\ N_{\text{rep}},\ \textit{proposed},\ V,\ V_{\text{old}}]$$

to their *halo view.* In such entry, the *updating* field is set to V_{old}, the identifier of the last view where the client was confirmed, i.e. the view from which the client information is now being updated. Nodes whose *halo view* already contained an entry for Id will now set the corresponding *updating* field to the current view identifier, so that the local entry reads

$$[Id,\ \textit{address},\ N_{\text{rep}},\ \textit{committed},\ V_{\text{old}},\ V].$$

Hence in this case, the *updating* field contains the view *to which* the information is being updated. Some extra modifications may be needed in case the local entry for Id was already $definitive$, or the confirmation round for Id was interrupted by a reconfiguration of the core group. Such cases will be separately described in the following sections.

When the updating round is completed, N_{rep} sets the client's local status to *committed* in view V, sends **Horizon**(Id, *V.members*) to the client and substitutes the token entry by **ConfirmTE**(N_{rep}, Id). When processing this entry, the remaining nodes will set their corresponding local entries for Id to *committed* in V.

5.4.5 Core Failures and Token Recovery

The basic description of the previous section assumes that no reconfiguration of the core group takes place during a given token round. In the most general case, nevertheless, such changes may occur, although they will be rare compared to the rate of client additions and disconnections. Token processing must be suspended whenever a change happens to the core view, as the HaloMS protocol may only run within a `committed` majority view. Thus all core nodes will stop the HaloMS protocol as soon as they have evidence of a change in the group, while keeping a record of the ring composition to be used during recovery phase. If the underlying core membership service is the HMS protocol, described in Chapter 4 or any other membership service that notifies temptative view installations, the HaloMs protocol may be stopped as soon as a membership event notifies a new view trying to be installed.

The pause implies that the token will no longer be sent until a new majority view is confirmed, in which the token must be regenerated. If conversely a minority group arises after reconfiguration, each involved member will clear its *halo view*. In order to maintain the consistency of the information about clients, such regeneration can only be carried on by the surviving node that was the last to receive the token before the change occurred and has therefore the most recent information.

When the HaloMS module at a core member receives the notification of an installed majority view, V_{M}, from its underlying core membership service, it sends a message to each of its represented clients informing them about those failed core members they have to exclude now from their respective *membership horizons*. If some core node has failed, its represented clients are *inherited* by the survivors. The substitute representatives are calculated deterministically by applying some predetermined rules, and are then in charge of notifying clients about their representative failure — and the others.

If the underlying membership service notifies also about the completion of view confirmation (as HMS may notify the `release` of a view), HaloMS may wait for such an event before entering the recovery phase, since at that moment all the participants of the new ring will have installed V_{M}. Otherwise, a core member may start the recovery when the confirmation of a majority view is perceived, although retransmissions or storage of the recovery token will be required as not all nodes will perform the view installation simultaneously.

When the node enters the recovery phase, it sends a **RToken** message to its most immediate predecessor from the former ring that survives in the currently installed view. The **RToken** contains the identifier of the sending node and a copy of its last received regular token.

The receiver compares the identifier of this recovery token, T_{rec}, with that of its own last received token, T_{last}. Then one and only one of the following scenarios occur.

- For exactly one node, N_{b}, the received identifier is older than the local copy, $T_{\mathrm{rec}} < T_{\mathrm{last}}$.

- For one single node, N_{b}, $T_{\mathrm{rec}} = T_{\mathrm{last}}$ and N_{b} follows (in the former ring order) the node that has sent the recovery token T_{rec}. This would happen in case the generator, i.e. node that increases the token identifier, has failed and the new token has not reached any of the surviving nodes.

In any of these cases, N_{b} can be identified as the boundary of the token advance prior to the reconfiguration. This node will thus be in charge of regenerating the token and its identifier, so that the counter is increased and N_{b} appears as the initial node of this incarnation and takes the role of token generator. Once the regular token is regenerated, N_{b} will send it again along the new ring. To regenerate the token, N_{b} must examine its own *halo view* and proceed according to the following rules.

1. For each client Id which is locally *proposed* with field $V = V_{\text{old}}$, information on the corresponding entry must be sought in the recovery token received from the next node.

   - If the recovery token contained the **ProposeTE**(Id) entry, [4] then the proposal round can be taken as complete, as all surviving nodes from the view in which the addition was proposed have seen this entry. A confirmation round may start now, but the round will also be used for updating. This is achieved by adding to the token an entry

     UpdateTE(N_{b}, Id, *address*, N_{rep}, V_{old}, V_{M}),

     where the representative N_{rep} is the one stored in N_{b}'s local entry for Id, if still alive, or its substitute otherwise. Before sending the token along the ring, N_{b} will also change its local entry to

     [Id, *address*, N_{rep}, *committed*, V_{old}, V_{M}],

     where the current view identifier, VM, is stored in the V_{u} field (see sect. 5.4.2).

   - If no entry for Id was present in the recovery token, then the proposal round did not finish, so that a new proposal must be now launched. N_{b} will thus add the entry

     ProposeTE(N_{b}, Id, *address*, N_{rep})

     to the token, with N_{rep} calculated as in the previous case.

2. If a client is locally *proposed* in $V = V_{\text{old}}$, but with a non-null value for the V_{u} field, say V_{old_2} (necessarily $V_{\text{old}_2} < V_{\text{old}}$), different cases may be found when examining the recovery token.

[4]For clarity, only the entry fields that are relevant for the discussion are explicitly shown in this section.

- If the **UpdateTE**(ID, *address*, N'_{rep}, V_{old_2}, V_{old}) entry was present in the recovery token, this means that the updating round is complete, and confirmation in V_{old} is due. Nevertheless, it will be performed by a new updating round, with

  $$\textbf{UpdateTE}(N_{\text{b}},\ Id,\ \textit{address},\ N_{\text{rep}},\ V_{\text{old}},\ V_{\text{M}})$$

  in the token. This indicates to the remaining nodes that Id was included by all nodes in view V_{old}, from which it is now being updated to V_{M}. Before passing on the token, N_{b} will also modify the local entry to

  $$[Id,\ \textit{address},\ N_{\text{rep}},\ \textit{committed},\ V_{\text{old}},\ V_{\text{M}}],$$

  being V_{M} the current view, to which the client's *horizon* is to be updated.

- In any other case, i.e. if the recovery token contained no entry about Id, or if it contained a **ProposeTE**, **ConfirmTE** or **DefinitiveTE** entry regarding this client, [5] or an **UpdateTE** referring a previously confirmed view different from V_{old}, then

  $$\textbf{UpdateTE}(N_{\text{b}},\ Id,\ \textit{address},\ N_{\text{rep}},\ V_{\text{old}_2},\ V_{\text{M}})$$

  is included in the token.

3. For each *committed* client with $V = V_{\text{old}}$ for which the V_{u} field is not set, the recovery token is searched for an Id entry.

   - If the recovery token contains **ConfirmTE**(Id), the confirmation round was completed and then

     $$\textbf{DefinitiveTE}(N_{\text{b}},\ Id)$$

     is included in the new token.

   - Otherwise, conclusion of the confirmation round must be ensured, using also the round for updating. So

     $$\textbf{UpdateTE}(N_{\text{b}},\ Id,\ \textit{address},\ N_{\text{rep}},\ V_{\text{old}},\ V_{\text{M}})$$

     is added to the token.

4. If a client Id was locally *committed* in $V = V_{\text{old}}$ with the corresponding V_{u} field value set to V', that means that an **UpdateTE** round confirming the client in V_{old} and updating to V' was in progress. Thus necessarily $V' > V_{\text{old}}$, and the interruption happened as the client, whose identity was confirmed in V_{old}, was being updated to V'. Whether the round completed or not must be figured out from the recovery token received from the next ring member.

[5]This would mean that the *recovery* token corresponded to a round regarding confirmation of Id in view V_{old_2}, before the updating started.

- If the recovery token contained **UpdateTE**(Id, $address$, N'_{rep}, V_{old}, V'), then the round finished, and confirmation regarding V' is due. As in the previous cases, it will be carried out by a new updating round. So,

  $$\textbf{UpdateTE}(N_{\mathrm{b}},\ Id,\ address,\ N'_r,\ V',\ V_{\mathrm{M}})$$

  is included in the new token.

- In any other case, the updating round did not finish, so

  $$\textbf{UpdateTE}(N_{\mathrm{b}},\ Id,\ address,\ N'_{\mathrm{rep}},\ V_{\mathrm{old}},\ V_{\mathrm{M}})$$

  is included in the token to retry it, directly updating *halo views* from V_{old} to V_{M}.

5. If the client Id is locally *removed*, then Id's representative had started its elimination.

   - If the recovery token contains **RemoveTE**(Id), the round was closed, so N_{b} will now include **DefinitiveTE**(N_{b}, Id).
   - Otherwise, the removal order is included again, as entry **RemoveTE**(N_{b}, Id).

After composing the token in the way described above, N_{b} must set the next available client identifier and the new token identifier. The field Cid of the regenerated token is set to

$$max(Cid',\ \{Id_{\mathrm{i}} \in halo\,view / status(Id_{\mathrm{i}}) = proposed\}) + 1,$$

being Cid' the corresponding field of the last token received by N_{b}. The identifier of the new token will be obtained by increasing the one in the last received token and replacing the old token generator by by N_{b}.

For as long as the core view V_{M} remains unchanged, N_{b} will hold the role of token generator. Each time it gets back the token, it will increase its identifier, indicating a new round was completed.

5.4.6 Token Processing

To complete the description of the protocol operation, we must now analyse in detail the operations performed by each core node every time it processes a new token, in order to maintain and update its *halo view* with the circulating information.

When a core node N receives the token from the precedent ring member, it saves a copy for recovery purposes and processes its contents before passing it on. For each token entry it is checked whether this node was the initiator of the corresponding round, in which case the token entry must be promoted to the next action. Otherwise, the information in the token entry is used to update the local *halo view* and the same entry is passed on to the following node in the ring.

The following rules describe how individual token entries are processed by a node N.

1. **ProposeTE**(N_{ini}, Id, $address$, N_{rep})
   - If $N \neq N_{\text{ini}}$, the following entry is created and inserted in the *halo view*,
   $$[Id,\ address,\ N_{\text{rep}},\ proposed,\ V],$$
   where V is the current view identifier.
   - If $N = N_{\text{ini}}$, then the local entry status is changed from *proposed* to *committed*, and the token entry is replaced by **ConfirmTE**(N, Id). Moreover, if $N = N_{\text{rep}}$, **Horizon**(Id, $V.members$) will be sent to the client upon local commit.
2. **ConfirmTE**(N_{ini}, Id)
   - If $N \neq N_{\text{ini}}$, the local value of *status* for client Id is changed from *proposed* to *committed*. If N is the representative for Id, as recorded by the local *halo view* entry, then **Horizon**(Id, $V.members$) is sent to the client. If the local entry held a non-empty V_{u} field, it is now cleared.
   - If $N = N_{\text{ini}}$, as the round initiator N changes the local *status* for Id to $definitive$, clearing the V_{u} field, if any. The token entry is also substituted by **DefinitiveTE**(N, Id).
3. **UpdateTE**(N_{ini}, id, $address$, N_{rep}, V_{old}, V)
   - If N's *halo view* does not contain an entry for Id, the following is now created and inserted,
   $$[Id,\ address,\ N_{\text{rep}},\ proposed,\ V,\ V_{\text{old}}].$$
   - If an entry for Id already existed, and $N \neq N_{\text{ini}}$, two cases must be distinguished.
     (a) If the local entry of Id is of the form
     $$[Id,\ address,\ N_{\text{rep}},\ definitive,\ V_{\text{old}},\ V_{\text{u}}],$$
     with or without a value of the updating field V_u , then it is now changed to
     $$[Id,\ address,\ N_r,\ definitive,\ V_{\text{old}},\ V],$$
     thus indicating that the addition was confirmed in V_{old} and its update to the current view V is now being proposed.

   (b) In any other case the local entry is changed to

$$[Id,\ address,\ N_{\text{rep}},\ committed,\ V_{\text{old}},\ V].$$

- Finally, if $N = N_{\text{ini}}$, the local entry is changed to

$$[Id,\ address,\ N_{\text{rep}},\ committed,\ V],$$

  and the token entry is replaced by **ConfirmTE**(N, Id).

4. **RemoveTE**(N_{ini}, Id)

   - If $N \neq N_{\text{ini}}$, then the local *status* value for the corresponding entry is changed to *removed*, provided N knew about Id. The entry is ignored if N had no local entry for client Id.
   - If $N = N_{\text{ini}}$, the local entry, whose *status* was already *removed*, is deleted from the *halo view*, and the token entry is promoted to **DefinitiveTE**(N, Id).

5. **DefinitiveTE**(N_{ini}, Id)

   - If $N \neq N_{\text{ini}}$, and there was no Id entry in the local *halo view*, this entry is ignored. If the client was known, then
     - if $status = committed$, then it is changed to $definitive$, and the corresponding V_{u} field is reset.
     - if $status = removed$, then entry Id is deleted from the *halo view*.
   - If $N = N_{\text{ini}}$, then this entry is eliminated from the token. If there is no entry for Id in the local *halo view*, this **DefinitiveTE** is signalling the end of the removal phase. The identifier Id is thus invalidated and will not appear any longer in the group.

After processing the information carried by the token, N will compose and add to the token any new entries that may need to be included. These entries can be of two types.

1. Proposals for new detected clients that N is ready to accept, by means of the procedure described in sect. 5.4.4.

2. Updates of N's represented clients. The decision about whether to launch an **UpdateTE** round regarding a certain client can be forced by the recovery procedure described in the previous section. That procedure does not affect client entries whose *status* is already *committed* along the whole ring, but only ongoing rounds at the moment of the view change. Nevertheless, if may be convenient to launch an update on some client if it has been connected for long enough across

several core changes. An heuristic algorithm can be used to decide on launching such rounds, taking into account the activity of the system, the frequency of core changes and the mean life of client connections. In any case, when such algorithm decides an update is due on a given client, an **UpdateTE** entry will be composed and inserted in the token by N.

If any **ProposeTE** entry is added, the *Cid* field in the token will be conveniently increased, as described in 5.4.2. Then the token can be passed on to the next ring member.

After finishing the token processing, with the corresponding update of the *halo view* the HaloMS protocol will notify upper components that have registered as `IMembershipListeners` about membership events regarding the changes occurred to the local view of the halo group. In particular, client additions (when they are confirmed) and client removals (when labelled *removed*) will be notified after each token pass.

5.4.7 Majority Recovery

In the previous sections the operation of HaloMS has been described regarding basic operation and recovery after a change of the core group. In the latter situation it was assumed that a majority core view survived through view changes, so that there was a token to recover from the immediately preceding view.

Nevertheless, a singular situation may arise when the underlying core group is partitionable and the majority temporarily disappears due to partitions or failures. In such case, the token will eventually disappear from the group, as the HaloMS protocol stops in minority groups, and at the moment of installing a new majority view, all members will have an empty *halo view* and no last token no initiate the recovery phase. In this situation, the usual procedure for determining the boundary of token advance described in 5.4.5 will not work, and some specific actions are required in order to name a new token generator that restarts the HaloMS protocol. In fact, a particular case of this situation will happen at startup, when the majority group is constituted for the first time and the initial token must be launched. Moreover, for a node that was in a minority group, the situation of lost majority is not distinguishable from joining a running majority group, so that the same procedure must be tried every time a node enters a majority group.

To solve this problem, the token recovery procedure must be slightly modified. Thus, when a node N_{min} installs a majority core view, if it had no previous knowledge of a *halo view* (for it came from a minority partition), and only if it is the member with the lowest node identifier in the group,[6] it will send a special empty **RToken** to its

[6]Any other deterministic criterion can be used, of course, to choose a single sender among the

immediate predecesor in the logical ring that corresponds to the current view. When a node receives such empty **RToken**,

1. if it came from a majority and had information about a last token, and a *halo view*, it will simply ignore the empty recovery token;
2. if it comes also from a minority, it sends an empty **RToken** to its own predecessor.

If the initial sender of the empty **RToken**, N_{min}, receives it back, then all members are coming from a minority, and none of them had information to recover the token. Thus N_{min} becomes the token generator for the current view, and restarts the HaloMS operation by constructing and sending a fresh token with identifier $Tid = 1$.

Notice that if the underlying core membership service provides notification of all majority views in the group history, as granted by the *Uniform Majority View Agreement* property (GM.6) discussed in 4.5, every node is able to know at the moment of confirming a majority view, whether they were part of the last existing majority, say V_{M}. Only those that were members of V_{M} need to be involved in the recovery.

In such situation, the HaloMS protocol might register as `IMembershipListener` of the corresponding events, and the basic recovery procedure described in 5.4.5 suffices to recover the token when a majority persisted.

members of the current view.

Chapter 6

Membership On–Demand

This chapter presents a specification of the group membership service for a different dynamic scenario, inspired by ad hoc networks. The On Demand membership service presented here [75, 76] is nevertheless applicable to more general environments. In particular it will be useful when reducing the load of reconfigurations is necessary, while strong membership semantics is required. We also present a detailed protocol, called MODUS, that implements the specification based on any existing basic membership service. MODUS performance characterisation is also described.

6.1 Mobile Ad Hoc Networks

One of the most outstanding cases among dynamic distributed systems is that of mobile ad hoc networks or MANETs. MANETs are dynamically changing systems formed by wireless nodes, whose nature may vary from very small sensors, as intelligent dust, to handheld devices, like PDAs or laptops, or diverse vehicle equipment. The connectivity graph of the network ay change with time as nodes move, thus conforming an *ad hoc* topology.

The spreading of wireless technology, together with a broad range of potential applications have conferred increasing importance to ad hoc networks. This trend is patent in the ongoing efforts to develop specific services for these systems, such as efficient routing, in the concerns about security issues, and in the number of specialised publications on the subject appeared during the last years.

The Internet Engineering Task Force (IETF) MANET Working Group [77] mentions the following distinguishing features of MANETs.

1. The topology changes dynamically. Since the nodes may move, the network topology is not fixed, but will, in general, change over time. Moreover, links can

be either unidirectional or bidirectional.

2. Links have variable capacity and constrained bandwidth. The characteristics of wireless communications, such as fading, noise or interference, contribute to reduce the transmission rate. As a consequence of the reduced capacity of wireless links, congestion occurs frequently.

3. Energy–constrained operation. Since mobile nodes are usually powered by batteries, their energy supply is finite, and energy conservation becomes the crucial aspect in system design.

4. Physical security is limited. Mobile wireless communications are more vulnerable to a number of security threats than wired networks. Eavesdropping, spoofing and denial–of–service attacks are more likely and have to be taken into account. On the other hand, the decentralised character of ad hoc networks also contribute to make them more robust against single points of failure.

These features make the behaviour of ad hoc networks different to that of wired systems, and must thus be taken into account when designing efficient protocols for this kind of systems. Together with the above–mentioned properties, scalability may also pose a challenge to the development of certain systems over MANETs.

The potential uses of MANETs include military, industrial and commercial applications, ranging from providing connectivity when conventional networks are not available, in emergency situations or rough terrains, to the support of cooperative work. Further applications may involve hybrid networks that include both fixed and mobile members. The field is under very active development, and new uses are likely to arise that may become more important than the ones envisioned at present.

In any case, applications for ad hoc networks will require the development of specific services that support them. The fundamental service on which other components may be built is routing. At present, many different routing protocols have been proposed [78] and studied, and their specifications are now standardised. More recent work on the field is centred on security issues.

Group communication services are also among the most useful tools for the development of distributed applications. In wireless environments, research activity concerning group communication is mainly devoted to providing efficient broadcast of data through the whole network. The guarantees attained by the proposed services are far from being as strict as their counterparts in wired systems.

If distributed applications with strong semantics, such as replicated services, are to be deployed in MANETs, stronger support will probably be required from the group communication services or other distributed components.

The inherent difficulties related to the achievement of strong guarantees in a distributed system, of which the impossibility of reaching consensus in an asynchronous system is paradigmatic, may become even harder in a MANET. In practical settings for fixed–topology systems those difficulties are usually overcome by making appropriate assumptions, based on knowledge about the system behaviour or configuration. In a MANET, however, higher unpredictability and dynamism may render such kind of assumptions not applicable at all. Some of the traditional distributed problems may not even be well–defined in the system conformed by an ad hoc network.

Here we are mainly concerned about the definition and solution of the membership problem. Having a membership service that provides well–defined semantics in the context of an ad hoc network is basic, as in any distributed system, for other components to achieve their required guarantees. Thus the development of group communication systems, fault tolerant distributed object support systems, highly available database middleware components and, in general, of any higher level application can benefit from the information and the semantics provided by a membership service.

Just like in wired systems, where the traditional specification of the membership service is not equally suitable for all types of systems and applications, the distinguishing features and requirements of MANETs raise the need for specific service and protocol designs. Moreover, the term MANET applies to a huge variety of configurations, depending on the scale of the network, the rate of changes, the mobility of nodes and other parameters. Depending on the particular scenario under consideration, the guarantees provided by potential applications, and thus the semantics they will require from the underlying services may differ.

6.2 The System Model

Many distributed applications designed for ad hoc networks are not extremely demanding regarding consistency and can do without a strict membership service. It is the case of best-effort or probabilistic flooding of data, or some components designed for peer–to–peer architectures. We aim to support the deployment of distributed applications with strong consistency requirements, as any replicated service. Thus our goal is to provide the strongest membership semantics, so to enable the support for high availability and fault tolerance also in the framework of ad hoc networks. Not all settings among the variety of MANET configurations are capable to or interested in running such applications.

One typical working setting for us would be a set of users, holding laptop computers or PDAs, that run a collaborative work application for which data consistency is necessary at least during certain periods of time. Therefore, we will focus on systems composed of up to few hundreds of nodes, so that the size of the network is reduced enough to allow some collaboration among all the participants. Besides, the identities and number of the network members may or may be not known in advance.[1]

The network topology is allowed to change dynamically, due to several factors.

1. Crashes will occur when a host suffers a power cut or fails due to any other reason.

2. Announced disconnections may also take place when some node voluntarily leaves.

3. Finally, the network may grow as a consequence of the appearance of new members or the recovery of failed ones.

This kind of changes will be relatively frequent, their absolute rate depending on the system size. We consider that a reasonable order of magnitude will be of $10 - 50$ changes per hour in environments of $\mathcal{O}(10 - 100)$ nodes.

Mobility is indeed possible in a collaborative scenario as the one we are picturing. Nevertheless it will not play a decisive role in our problem, since we focus on a situation where collaborating nodes, i.e. those that take part of the group at a certain point, will mostly stay still as the collaboration takes place.

Failures may occur both to nodes and to communication channels. For nodes we consider the crash model. On the other hand, wireless communications are sensitive to shading, collisions, etc. This may produce link failures, as well as message losses, duplications or disordering.

The underlying transport service is assumed to be equipped with a routing service, that enables messages to be individually addressed to any member of the network. As discussed in the previous section, this is a reasonable assumption, as several standards and implementations of routing services exist for ad hoc networks. Neither reliability nor any particular order guarantees are assumed to be provided by the transport. The transport might also provide a broadcast primitive, that only ensures best–effort to make the message arrive to all members without atomicity guaranties, although that is not required. If such a service is not available, the membership component can implement it by itself, if required.

A typical occurrence of the described scenario will be the group of attendants to a conference or a similar event, that wish to execute some distributed application for collaborative work, and at a given point require agreement among a subset of nodes whose membership is maintained with the classical consistency guarantees.

[1]This point will be made clearer later.

6.3 On–Demand Membership Service

The maintenance of membership information with consistency guarantees within a dynamically changing group forcefully involves a sustained flow of messages among all the nodes in the network. Whether the membership service relies on a failure detector or not, if a minimum liveness of membership information is to be guaranteed, messages must be exchanged among members with certain periodicity in order to monitor possible failures or disconnections. Some group communication services, on the contrary, rely on messages exchanged by other components in order to avoid such forced flow of messages. Even in such cases, the appearance of new nodes (or reconnections of previously failed nodes), if occurring at an appreciable rate, will force frequent reconfigurations. Given the limited power of portable devices, this may turn out to be prohibitively costly for ad hoc networks.

Notice also that regular membership services require several message rounds in order to commit a view. In particular, it has been shown that membership can be solved in ≈ 3 communication rounds, if a solution based on consensus is adopted.[2] If joins happen very often, changes may happen faster than stabilisation of the membership set, so that depending on the dynamism of the network, running a standard membership monitor could become even unfeasible. In such cases, the strict consistency guarantees provided by a membership service may be impossible to achieve. Our interest is focused on an intermediate case, where the dynamism of changes is high enough to turn continuous running of membership support impractical, but not so high that consensus becomes impossible. In that situation, an application that makes use of the strong membership guarantees only for well–defined periods of time would benefit from a specific membership service.

With this in mind, we propose a specification of a membership service for ad hoc networks, and, in general, for any energy or bandwidth–aware environment. The main feature of the service is being *on–demand*, i.e., it can be turned on only when the interested applications require it, providing the strongest semantics for as long as required and avoiding unnecessary energy consumption when no upper component is asking for such consistency on any of the participant nodes.

6.3.1 Specification of the On–Demand Membership Service

The set of properties that specify a traditional membership service [22] ensure strict consistency of the membership information. Such consistency properties are most useful for higher components to implement their own semantics and reason about their correctness. We aim to control energy consumption without losing this advantage, therefore our approach is to provide equally consistent membership information, but

[2]See [60] and references therein.

only when some client application running on some of the participant nodes demands those guarantees. This will allow nodes to save energy and bandwidth consumption when no one is requesting membership services.

The On–Demand membership service is assumed to satisfy the general interfaces in figs. 3.1, 3.2 [79], corresponding to a client–server architecture for the membership service and its relationship with other components of the system.

The On–Demand Membership Service proposed here is specified as follows.

Property OD.1 (Consistent Membership)**.** *Any registered membership client will be provided with membership information with the strict consistency properties that define a regular membership service according to [22, 10], as explained in Sect. 3.1.2.*

The previous properties implies that, when the membership service is operating, it must fulfil the four safety properties.

1. Self Inclusion;
2. Initial View Event;
3. Local Monotonicity;
4. View Agreement.

The basic specification of a partitionable membership service requires also a liveness property, namely Membership Precision, which requires that, if there are stable components in the system, the same (correct) view is installed as the last one in every node of the same component. Nevertheless, in the case of a service On–Demand, even if a stable component exists, which is in any case unlikely in a dynamical environment, some members of the component might be not requiring membership services. In such case, the previous property would never be satisfiable. To solve this problem, we may substitute the term *stable component* in the enunciate of Membership Precision by *stable active component*, denoting a subset of clients (processes) that remain registered as membership listeners to correct, connected nodes.

Property OD.2 (Active Membership Precision)**.** *If there are stable active components in the system, the same (correct) view is installed as the last one in every process of the same component.*

As in the case of a traditional membership service (RMS), this property is conditioned to the survival of such components for a long enough time.

Finally, the property that allows the On–Demand service to save energy is enunciated as.

Property OD.3 (Eventual Stop)**.** *If all membership clients unregister, every node will eventually stop its local membership protocol.*

6.4 MODUS

In this section, a group membership protocol named MODUS is presented, which is designed to provide the semantics of the specification in the last section from a *regular membership service* (RMS). MODUS can thus be implemented on top of any existing membership protocol, provided the latter conforms to the generic interface in 3.1.

6.4.1 Architecture

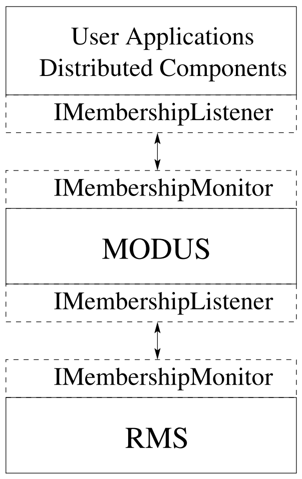


Figure 6.1: Basic architecture of MODUS service.

The architecture of this service is shown in fig. 6.1. Each potential member of the group is assumed to operate an instance of certain RMS protocol, which must ensure the properties of the standard specification of the membership problem through the generic interface `IMembershipMonitor`. No more assumptions are done on the RMS. Thus, MODUS does not face the question of whether the identities and number of nodes must be known in advance, neither it imposes the partitionable or non–partitionable character of RMS or the collective or individual startup.

On every node, MODUS will act as the only `IMembershipListener` of the RMS. It will also hold, in exclusive, the capability to turn the RMS on and off, by appropriately calling the `joinGroup` and `leaveGroup` methods. Any other component or application interested in membership events should register as listener of MODUS. Since this offers the generic interface of `IMembershipMonitor`, the application can make use of MODUS as if it was a regular membership service. Nevertheless, for the On–Demand policy to be advantageous, any membership listener must take care of registering right before and unregistering immediately after each period in which membership information is required.

The specification of MODUS as an Input/Output Automata is included in appendix B.1. Nevertheless, for a better understanding of the protocol operation, a less formal description in terms of states and transitions is presented in the following sections.

6.4.2 Basic Ingredients

The protocol is specified as a set of subalgorithms for the different states of MODUS. Transitions among them are driven by invocations from external membership listeners, by membership events generated by RMS, and by dedicated MODUS messages that allow the different participants to operate their local RMSs consistently. MODUS makes use of the distributed registration and unregistration of listeners to decide about the start and stop of the underlying RMS, so that when there is no registered user in the system listening to membership information, the regular membership protocol is stopped on all nodes.

Local State

The state maintained by MODUS at each node consists of the following.

- `localClients`: a list of references to all locally registered membership clients;
- `activeNodes`: a list of participant nodes which currently hold registered listeners.

The first one is local, whereas the second one plays the role of a distributed list of references holding the set of active nodes. Based on the information on this list, MODUS will decide when there are no registered membership listeners in the system and thus the underlying RMS must be stopped.

States

MODUS has four possible states, depicted in fig. 6.2.

- UNAVAILABLE: In this state, the underlying RMS is switched off, and the node is invisible to the membership group, i.e. it is not allowed to take part in any membership agreement.
- AVAILABLE: In this state, the node has no locally registered client and its RMS is switched off, but upon request of some other member the node can be forced to start it and take part of membership rounds.

- PASSIVE: The local RMS is running, but no client is registered, so that operation of the RMS, with its consequent energy consumption is enterely in benefit of other nodes.

- ACTIVE: There are locally registered membership clients, and the RMS is running and providing them with membership information.

Messages

The distributed operation of MODUS is allowed by the set of dedicated messages described below. Each of them carries specific arguments, as detailed in the list, and is also labelled, when required, with the view identifier of the RMS view.

- **ServiceReq()** is broadcast by any node that transitions from the AVAILABLE to the ACTIVE state due to the registration of a local client. It is used to require participation of other AVAILABLE nodes in the membership agreement that is to be achieved.

- **ActiveMsg(**N**,** Vid**,** Vid_{old}**)** is sent by the ACTIVE node N either to anounce its entering this state to all members of the group, or to update their information, after a change of view has occurred, maybe preventing the completion of a previous difussion of state. Only in the second case, Vid contains the view identifier of the current view, and Vid_{old} that of the former view.

- **PassiveMsg(**N**,** Vid**,** Vid_{old}**)** is used in the same way, to indicate transitions to PASSIVE state (or to update the same information).

- **ActiveList(**N**,** Vid**,** Vid_{old}**,** *list***)** is sent to all members of a just formed view, Vid, by just one of the survivors, N, of the former one, and only in case there are new joined nodes that where not present in the older view, Vid_{old}.

As already mentioned, no reliability is ensured by the underlying transport. Any message can then be lost, thus preventing the completion of the algorithm. This can be avoided by reliable point–to–point delivery, which can be explicitly implemented for each message. In order to ease the description and realisation of the protocol, however, a more general `rmcast` primitive is implemented as support for MODUS, as for other protocols in this thesis. The `rmcast` primitive guarantees reliable point–to–point delivery within a given RMS view by periodically resending messages until they are acknowledged or a membership change occurs.

That is the semantics of `rmcast` in the following description, whereas `broadcast(`m`)` will denote an attempt to flood the network with the message m, as best provided by the underlying transport.[3]

6.4.3 Protocol Operation

The subalgorithms for the different states of MODUS are shown in figs. 6.3-6.6. Here we present an overall description of the protocol operation.

MODUS starts in the UNAVAILABLE state, with the RMS switched off, and it cannot be forced to take part in resource–consuming membership rounds. Applications are also not allowed to register as membership listeners in this state. Before this is possible, `MODUS.joinGroup` must be invoked.

The node becomes then AVAILABLE, but the RMS is not immediately started. A client application C_a may register now as `IMembershipListener` by an invocation to `MODUS.registerListener(`*evt*, C_a`)` for some type of membership event *evt* in the set of events notified by the RMS. MODUS will then add the application to `localClients`, invoke `RMS.joinGroup` to start its local RMS, move to ACTIVE state and request other nodes to participate in the membership agreement by broadcasting a **ServiceReq** message. Other AVAILABLE nodes that receive such message will also switch on their local RMSs and move to the PASSIVE state, whereas UNAVAILABLE nodes will simply ignore the message.

The node N that is transitioning to the ACTIVE state will also add its own identifier to the list of active nodes and send a message **ActiveMsg(**N**)** to all members of its current view. This message contains the node identifier, but no view information. Any ACTIVE or PASSIVE members that receive the **ActiveMsg** will update their own versions of the list and include N in them, provided they are in a view that contains N.

For as long as a client application C_a is registered as listener of MODUS, the protocol will notify it of membership events whose type matches the one in C_a registration. The reference to C_a is kept in the `localClients` list until the `unregisterListener(`*evt*, C`)` method has been called for all the *evt* values for which `registerListener` was previously invoked. When the last unregistration takes place, the reference to C_a is removed from the local list. Once the `localClients` list of a node N which was in the ACTIVE state is empty, N removes its own identifier from the local instance of the `activeNodes` list, it *rmcasts* a message **PassiveMsg(**N**)** to all members of the current RMS view and moves to PASSIVE state. The receivers of the message will also eliminate N from its local list `activeNodes`.

A node in the PASSIVE state has no locally registered client, but runs its local RMS for the benefit of other requesting members. It will stay in this state until a local listener

[3]If no such facility is provided by the transport, the flood can be substituted by a single emission within the radio range.

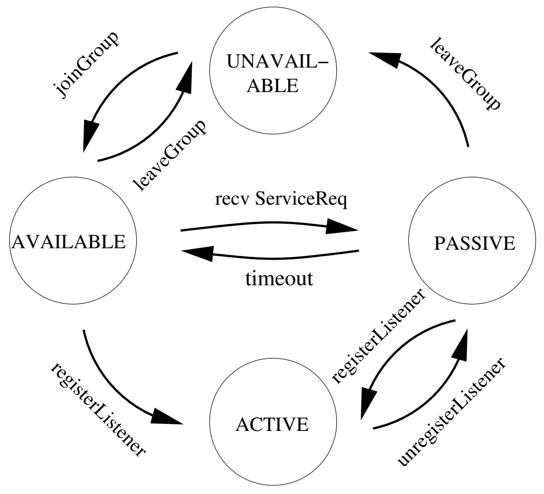


Figure 6.2: States of MODUS protocol, with events and messages driving transitions.

registers, promoting the state to ACTIVE, or until the list of active nodes is empty, in which case it may turn off its RMS and go back to the AVAILABLE state. This transition does not take place immediately, but a timeout is established to allow information on active nodes to reach passive members of a just formed view. Otherwise, a single requesting node would, with high probability, end up receiving simply a membership view with itself as the only member.

By invoking `MODUS.leaveGroup` any node with no registered clients, in the PASSIVE or AVAILABLE state, is forced to leave any membership group it took part of, irrespective of the presence of other active members, and enters the UNAVAILABLE state.

Interaction with RMS View Changes

MODUS intercepts all membership notifications of RMS, in order to pass them to the proper clients and to make use of RMS semantics to simplify its own operation. In particular, MODUS itself is only sensitive to installation of new views.

After `RMS.joinGroup()` has been called, the **Initial view event** property of RMS ensures that MODUS will see the installation of a membership view. Depending of the characteristics of RMS (individual or collective startup) such view may be a singleton initial view, or a real image of the group. In any case, that provides MODUS with the destination set for the `rmcast` of its initial **ActiveMsg**. From that point on, when RMS notifies a view change, two situations may occur.

If some nodes are missing from the new view, with respect to the former one, MODUS removes failed nodes from the `activeNodes` list.

If the new view, V_2, contains new nodes, not present in the former one, V_1, they have to be updated about the state of older members. To this end, one of the survivors, N_s, is chosen to broadcast the active list to all members in a message **ActiveList(**N_s, V_2, V_1, `activeNodes`**)**. The deterministic election of the sender is trivial given the RMS view guarantees. Other V_1 survivors wait for the message. If the received list does not reflect correctly the status of a given node N_r, as may happen if changes take place during activation or deactivation of the node, it will `rmcast` an **ActiveMsg(**N_r, V_1, V_2**)** or **PassiveMsg(**N_r, V_1, V_2**)** referring both the current view and the view from which the node is a survivor.

Since the underlying transport offers no guarantee with regards the order in which messages are delivered to their destinations, **ActiveMsg** and **PassiveMsg** may be received before the **ActiveList** they follow. MODUS exploits the guarantees regarding consistency and order of RMS views to label and process some messages in the proper order.

In order to minimise the need to resend messages that are received before they can be processed, instead of dismissing them, early **ActiveMsg** or **PassiveMsg** received with reference to a view V_1 are stored until V_1 has been installed and the active list corresponding to V_2 has been processed, or until the installation of a subsequent view discards them.[4]

```
algorithm MODUS
type
   state_t = { UNAVAILABLE,
        AVAILABLE, PASSIVE,
        ACTIVE };
var
   state : state_t;
   activeNodes : list of
             nodes;
   localClients : list of
             listeners;
   view : RMS mbship group;
begin
   state := INACTIVE;
   activeNodes := null;
   localClients := null;
   view := null;
   while true do
      case state of
      UNAVAILABLE: unavailable;
      AVAILABLE:   available;
      PASSIVE:     passive;
      ACTIVE:      active;
      esac
end
```

Figure 6.3: Basic algorithm for MODUS.

[4]For simplicity, these procedures are not explicit in the schematic algorithms of figs. 6.3-6.6.

```
algorithm unavailable
begin
   wait for event
   case event of
   joinGroup:
      state := AVAILABLE;
   esac
end

algorithm available
var
begin
   wait for event
   case event of
   recv(ServiceReq):
     RMS.joinGroup();
     state := PASSIVE;
   registerListener:
      localClients.add(listener);
      activeNodes.add(thisNode);
      broadcast(ServiceReq);
      RMS.joinGroup();
      state := ACTIVE;
   leaveGroup:
      state := UNAVAILABLE;
   esac
end
```

Figure 6.4: Algorithms for the unavailable and available states.

```
algorithm passive
var
  timer : integer;
  view_p : RMS mbship group;
init
  timer := -1;
  view_p := null;
begin
  if activeNodes is empty
         timer=WAIT_FOR_ACTIVE;
  wait for event
  case event of
  recv(ActiveMsg):
    activeNodes.update(ActiveMsg);
    if (msg(view,view_p) and
      outdated info on thisNode)
        rmcast current status as
          ActiveMsg/PassiveMsg
          wrt (view,view_p);
    if activeNodes not empty
        timer := -1;
  recv(PassiveMsg):
    activeNodes.remove(sender);
    if activeNodes is empty
          timer=WAIT_FOR_ACTIVE;
  notify(view_event):
    view_p := view;
    view := view_event;
    activeNodes.remove(failed);
    elect leader among
       view_p survivors;
    if (joined nodes and
        leader=thisNode)
          rmcast(ActiveMsg)
           with activeNodes
           wrt (view,view_p);
    if activeNodes is empty
          timer=WAIT_FOR_ACTIVE;
  registerListener(listener):
    localClients.add(listener);
    activeNodes.add(thisNode);
    state := ACTIVE;
  timer timeout:
    RMS.leaveGroup();
    state := AVAILABLE;
  leaveGroup:
    RMS.leaveGroup();
    state := UNAVAILABLE;
  esac
end
```

Figure 6.5: Algorithm for the passive state.

```
algorithm active
var
  view_p : RMS mbship group;
init
  view_p := null;
begin
  rmcast(ActiveMsg);
  wait for event
  case event of
   recv(ActiveMsg):
     activeNodes.update(ActiveMsg);
     if (msg(view,view_p) and
       outdated info on thisNode)
         rmcast ActiveMsg
           wrt (view,view_p);
   recv(PassiveMsg):
     activeNodes.remove(sender);
   notify(event):
     if event is a view_event
        view_p := view;
        view := view_event;
     activeNodes.remove(failed);
     elect leader among
        view_p survivors;
     if (joined nodes and
         leader=thisNode)
           rmcast(ActiveMsg)
            with activeNodes
            wrt (view,view_p);
     localClients.notify(event);
   registerListener(listener):
     localClients.add(listener);
   unregisterListener(listener):
     localClients.remove(listener);
     if localClients is empty
        activeNodes.remove(thisNode);
        rmcast(PassiveMsg);
        state := PASSIVE;
   esac
end
```

Figure 6.6: Algorithm for the active state.

6.4.4 Analysis and Performance

The main advantage of the on–demand approach with respect to a RMS is the potential energy saving when the membership information is not in use for significant periods of time.

If membership information is required in a continuous way, running MODUS on top of a RMS implies an overhead in terms of computing power and number of sent messages, and thus an extra cost in energy and bandwidth. The implementation of MODUS starts being worthwhile when the time RMS is disconnected compensates the overhead during periods of activity. This circumstance depends on the multiple parameters that control the performance of the algorithms.

The evaluation of distributed algorithms is currently an open question [80]. The complexity of distributed protocols and their dependence on the particularities of the system on which they are deployed, make it difficult to characterise their performance in an absolute manner.

In this section we are interested in a comparison of the energy spent by MODUS and RMS in terms that are as general as possible. The absolute energy consumption of either protocol depends on the features of the physical devices that conform the network, such as their characteristic sending, receiving and computational powers. It also depends on the particular configuration of the network, which affects how messages are routed, retransmitted, etc. The state of the network at a given instant will determine the occurrence of collisions, interferences, and message losses, and will thus have also an impact on consumption. These network factors are variable, specially in the case of an ad hoc network.

Therefore, we focus on a magnitude that is mostly protocol–dependent, namely the minimum number of messages sent per node and unit time required by each protocol's operation. Every sent message implies a certain power consumption. Energy will also be spent in reception, but the number of received messages is proportional to that of messages sent, so that both quantities are not independent. Thus there will be an unavoidable rate of energy consumption associated with MODUS or RMS, and proportional to the rate of sent messages. Notice also that we are not taking into account the energy consumption due to overhearing, which may be significant in wireless environments.

Although not determining the absolute energy consumption of the protocols, the comparison of the analysed quantities may give us a hint of the relation between the energy spent by each of them as a function of the multiple problem variables. We may use this sent message rate to characterise the behaviour of MODUS under different scenarios and then validate the model by comparing it to measurements.

Relevant Parameters

Among the large number of parameters that affect MODUS performance, we are mainly interested in studying the effect of varying the following.

- Size of the system, measured by the maximum number of nodes that conform the cluster, N.

- Patterns of registration/unregistration of clients. Since the number of clients registered as membership listeners in one node is only local, the only relevant information is the time of the first registration and the last unregistration, thus we may consider a single client application per node. We will call τ_r (τ_u) the average frequency of registration (unregistration) of the client.

- Percentage of active nodes. The application may be only running on a fraction of nodes in the system, represented by the parameter $F \in [0, 1]$.

- Pattern of failures/rejoins. If nodes fail and reconnect at a high enough rate, the number of potential members in the group will be variable, and affect the cost of the protocol. We define τ_f (τ_j) to model the number of failures (reappearances) per node and unit time.

There is a number of additional parameters whose value determines the sending rates of MODUS, but are not included as variables in our model. Instead they are considered as external parameters.

- Average time required to complete a *rmcast* in the network, t_{rc}. Since no primitive is available, this can be estimated as the cost of sending a broadcast/multicast to all members and receiving all answers.

- Average time required by RMS to install a membership view, t_{view}. This is a parameter that depends on the particular RMS being used and on the characteristics of the network. If its value is not available, it can be estimated from the number of communication rounds required by the RMS in order to install a view.

Analysis

A typical client application could work according to the following pattern.

1. Most of the time, the application does not need agreement among the members of the network, and membership services are not required.

2. In order to perform some particular operations, certain level of agreement is required. Thus the application makes use of the information delivered by the membership service for the time such operations are being carried.

For the time no application, on any of the nodes, is requiring membership information, MODUS takes charge of switching off the RMS, and the message consumption is reduced in exactly the rate of RMS. This includes possible control messages forced by the RMS or a failure detector in order to keep liveness of the membership information, but also the necessary rounds to change views as new members join the group. The former flow of messages may be not present in the case of a membership service that does not rely on its own control messages to ensure liveness, but reacts to other components. The latter, on the contrary, can only be avoided if the RMS is switched off.

During the time of membership agreement, MODUS imposes an overhead in the number of exchanged messages. Therefore, after a time of operation the total consumption will depend on the ratio of membership activity to membership rest time. Let τ_{MODUS} be the average sending rate per node of MODUS for the periods of membership activity, τ_{RMS} the average sending rate of RMS along all its execution, and γ_{act} the time for which membership information is required. Then after a total running time T, MODUS will have exchanged less messages, and thus enabled some energy and resources saving, if

$$\tau_{\text{MODUS}}\gamma_{\text{act}}T \leq \tau_{\text{RMS}}T \Leftrightarrow \gamma_{\text{act}} \leq \frac{\tau_{\text{RMS}}}{\tau_{\text{MODUS}}}. \tag{6.1}$$

The message consumtpion of RMS is determined by the particular membership service in use. Even if the particular RMS does not involve specific monitoring messages, changes to the composition of the group due to appearances and disappearances will imply installation of views with the subsequent message rounds. Every appearance or disappearance of a node will imply a change of view. Since the cost of a view installation is lower–bounded by the cost of consensus plus one communication round [60], if the system suffers n_{app} (n_{disapp}) node appearances (disappearances) per unit time, we may write

$$\tau_{\text{RMS}} = \tau_{\text{monitor}} + (n_{\text{app}} + n_{\text{disapp}})\,\frac{3C_{\text{round}}(N)}{N}. \tag{6.2}$$

Eq. 6.1 gives us an upper bound to the fraction of time of membership activity if MODUS is to produce some saving. The value of that ratio will vary depending on the activity pattern of the network and the client applications during the time of membership agreement.

Let us consider a client application which registers as a membership listener, stays connected for a short period of time, as it performs some definite operations and unregisters again. We model the patterns of registration and unregistration at each individual node with normal distributions of means μ_{r} (mean registration time) and μ_{u}

(unregistration time) and standard deviations σ_r and σ_u respectively. Then for the rate of connections and disconnections we have $\tau_\mathrm{r} = \tau_\mathrm{u} = \frac{1}{\mu_\mathrm{r}+\mu_\mathrm{u}}$. Moreover, the asymptotic probability of a given node to be registered at any instant of time is $P_\mathrm{reg} = \frac{\mu_\mathrm{r}}{\mu_\mathrm{r}+\mu_\mathrm{u}}$.

If no simultaneous unregistration of all nodes occur, during activity periods we have the basic RMS sending rate plus an extra `rmcast` each time a new node gets active or passive.

$$\tau_\mathrm{MODUS} = \tau_\mathrm{RMS} + (\tau_\mathrm{r} + \tau_\mathrm{u}) F C_\mathrm{rc}, \tag{6.3}$$

being C_rc the cost, in terms of number of messages, of completing a `rmcast`. Such cost will depend on the size of the group which, with no full unregistration, is kept by MODUS to the maximum, N. Besides sending the message to all nodes, `rmcast` implies receiving an answer from everyone. Thus we may estimate $C_\mathrm{rc} \propto 2N$.[5] The overhead of MODUS is then proportional to N, and governed by the frequency of listener registration–unregistration per node.

The rate τ_RMS depends on the particular RMS. If the RMS does not force monitoring messages, as we are only considering a situation in which no "hard" changes occur to the group, this rate will be null. Otherwise, its value will depend on the liveness imposed by RMS.

As registration periods become shorter and sparse, there may be a non-negligible probability that, even within activity times, all clients unregister at once, and MODUS forces RMS disconnection. In that case, the total number of messages sent by MODUS is reduced in the amount that corresponds to periods of deactivation, but in change there is an overhead due to the cost of reconfiguring the group from scratch. The exact number of messages exchanged when reforming the group and during disconnection are highly variable and may depend on a number of factors, such as the time RMS takes to install a new view, the succession of group views until the total membership is established, or the average time a `rmcast` operation takes to complete in the network.

Instead of trying to characterise exhaustively these costs, we make an analysis based on a worst and best case to set bounds to the sending rates.

We focus our analysis on the activity times, during which MODUS is at least AVAILABLE on every node. During such periods, we may schematically write the sending rate of MODUS as

$$\tau_\mathrm{MODUS} = \tau_\mathrm{cont}\left(1 - \frac{T_\mathrm{disconn}}{T}\right) + (\theta_\mathrm{conn} + \theta_\mathrm{disconn})\frac{\nu_\mathrm{disconn}}{N}, \tag{6.4}$$

being τ_cont the sending rate of MODUS (including the RMS component) as calculated above, $\frac{T_\mathrm{disconn}}{T}$ the fraction of the total time T for which there is no client connected on any of the nodes, $\theta_\mathrm{conn(disconn)}$ the overhead due to reconnecting (disconnecting) the

[5]In case we take into account possible message loss and resending, this cost may include an extra term which is a higher power of N.

whole group, and ν_{disconn} the frequency at which a simultaneous unregistration occurs. If the individual behaviour of the listeners is determined by $\mu_{r(u)}$, $\sigma_{r(u)}$, their effect on the sending rate will be included in the global parameters, ν_{disconn} and T_{disconn}.

The fraction of connection time depends heavily on the pattern of registration and unregistration of the client application. Even for our simple model it is not possible to deduce an analytical expression for ν_{disconn} and T_{disconn} in terms of $\mu_{r(u)}$, $\sigma_{r(u)}$, N and F. Nevertheless, a Monte Carlo simulation can be easily done so to estimate the relevant parameters in all the interesting cases.

The overhead θ_{conn} of composing the whole group after a global disconnection, once the first client registers again, depends on the operation of RMS. We may distinguish two components, one due to the extra messages exchanged by MODUS itself and the other due to RMS regular operation when forming a new group.

$$\theta_{\text{conn}} \equiv \theta_{\text{conn}}^{(\text{RMS})} + \theta_{\text{conn}}^{(\text{MODUS})}$$

The first node becoming active will broadcast a **ServiceReq** to all reachable nodes. Since no group was already formed, the receivers will then start joining a membership view, according to the particular RMS. The pattern of reconnection may vary from all nodes entering the same final view immediately, to a succession of views, each containing one more node, that will be installed if RMS lets nodes join one by one.

Each view that includes new nodes means an **ActiveList** message to update joined nodes about the state of the older ones. This implies at least two `rmcast` from the previously disjoint subgroups. Besides, if the view change took place while some node was `rmcasting` a change of state, and this operation had not finish, it is possible that such node needs to repeat the `rmcast` within the whole new group, after the first **ActiveList** message. The occurrence or not of such repetition will depend on different parameters, such as t_{bc} (the average time cost of a `rmcast` operation) and t_{view} (the average time it takes for RMS to install a new view), but also on non-deterministic factors, as the pattern of reconnection or the number of views the group goes through before reaching the full membership again, which are harder to quantify.

In order to analyse the behaviour of MODUS, we focus on two extreme cases, for which the minimum number of sent messages can be estimated.

- In the best possible case, all nodes will join at once. If the RMS uses collective startup, there will be only one view installed, whereas if individual startup is used, each member will first be in a singleton view and then enter the definitive, global view. The RMS will thus impose at least an overhead equivalent to the cost of installing a view. Thus the RMS overhead per reconnection will be

  $$(\theta_{\text{conn}}^{(\text{RMS})})_{\text{best}} \geq 3C_{\text{round}}(N) \approx 6C_{\text{bc}} \approx 6N.$$

  In this situation, MODUS will not need extra messages besides the continuum rate to update joined members, as no former view exists. $(\theta_{\text{conn}}^{(\text{MODUS})})_{\text{best}} \approx 0$.

- In the worst case, the RMS may force nodes to join the group one by one. In such situation, the RMS installs $N-1$ views of monotonically increasing size and thus
$$(\theta_{\text{conn}}^{(\text{RMS})})_{\text{worst}} \geq \sum_{k=2}^{N} 3C_{\text{round}}(k) \approx 3(N^2+N-2).$$
The overhead imposed by MODUS during the succession of views installed every time the group recomposes can be calculated as follows. Each view implies two new **ActiveList** messages among the members, plus the `rmcast` operations for each registration or unregistration of one of its participants, but also the repetition of the ones that did not finish before the view change. Assuming $t_{\text{view}} \gtrsim t_{\text{rc}}$, we may estimate that those changes that take place within $t_{\text{view}} - t_{\text{rc}}/2$ from the installation of the view have time to complete their `rmcast`, whereas the ones started during the last $t_{\text{rc}}/2$ will not finish. They will impose an overhead of only a fraction of the total cost C_{rc} and then a full new `rmcast` in the following view. All in all, the overhead per view of cardinality k, with $1 < k < N$ can be written as
$$4k + \left(t_{\text{view}} - \frac{t_{\text{rc}}}{2}\right)(\tau_{\text{r}} + \tau_{\text{u}})F\,2k^2 + \frac{t_{\text{rc}}}{2}(\tau_{\text{r}} + \tau_{\text{u}})F\,(3k^2 - 2k).$$
Thus, after summing over all views, and taking into account that the last one imposes only the overhead of completing delayed notifications, plus the pair of **ActiveList** messages, and including also the cost of the initial **ServiceReq** broadcast, we may write
$$\begin{aligned}(\theta_{\text{conn}}^{(\text{MODUS})})_{\text{worst}} \approx\; & 2N^2 + 3N - 2 \\ & +2\left(t_{\text{view}} - \frac{t_{\text{rc}}}{2}\right)(\tau_{\text{r}} + \tau_{\text{u}})F\left(\frac{N^3}{3} + \frac{N^2}{2} - \frac{5N}{6} - 1\right) \\ & +\frac{t_{\text{rc}}}{2}(\tau_{\text{r}} + \tau_{\text{u}})F\left(N^3 - \frac{5N^2}{2} + \frac{3N}{2} - 1\right). \qquad (6.5)\end{aligned}$$
Besides, in this situation, the time of recomposing the group, $\approx (N-2)t_{\text{view}}$, is not negligible, and thus the contribution of the continuum rate during such reconfiguration will be different from that of the full group situation. To account for this effect we may substitute T_{disconn} by an effective quantity $\tilde{T}_{\text{disconn}} \equiv T_{\text{disconn}} - \nu_{\text{disconn}} t_{\text{view}}\,(N-1)$ and add a contribution
$$\tau_{\text{cont}}^{(\text{RMS})} \frac{\nu_{\text{disconn}}}{N} t_{\text{view}} \left(N^2 - N - 2\right),$$
which would be the contribution to the continuum rate of RMS of those nodes that are already (remain) connected as the group gets reconstructed (decomposed).

The overhead of breaking up the group, once a global disconnection occurs, depends on the simultaneousness of all nodes noticing the absence of active members, and also on the operation of RMS. The pattern of disconnection may vary from all nodes switching off their RMS at once, with basically no overhead due to installation of intermediate views, i.e. $\left(\theta^{(\mathrm{RMS})}_{\mathrm{disconn}}\right)_{\mathrm{best}} \approx 0$ to a succession of views, each one containing one less node, that are installed as nodes switch off their RMS one by one, in which case $\left(\theta^{(\mathrm{RMS})}_{\mathrm{disconn}}\right)_{\mathrm{worst}} \gtrsim 3(N^2 + N - 2)$. Nevertheless, if these views do not include new nodes, but are monotonically decreasing, they will not impose diffusion of **ActiveList** messages, and thus the disconnection overhead due to MODUS is negligible in our approximation. An important exception, however, is the case when the succession of views that are installed as nodes notice the global disconnection does not finish before a new client registers, as this would hinder the switching off of all RMSs. In our model we account for this fact by reducing the effective inactivity time in an amount proportional to the average time cost of a `rmcast` operation, t_{rc}.[6] Otherwise, we neglect $\theta_{\mathrm{disconn}}$.

Experiments

The experimental evaluation of MODUS performance should include in principle different patterns of registration-unregistration of membership listeners (i.e., different percentages of active nodes and different rates of connection–disconnection), different patterns of appearance–disappearance of nodes, and different sizes of the system. In order to validate the model and extract some conclusions about the performance of MODUS in a practical setting, we have measured the average sending rates in a set of test scenarios where the parameters are varied over a certain range of values, defined as follows.

1. The size of the system is chosen as $N \in \{20, 40, 60, 80\}$ nodes.
2. The percentage of nodes which hold membership clients capable to register as membership listeners $F \in \{30, 60, 100\}\%$.
3. The pattern of registration–unregistration of clients is varied over a finite set of patterns, defined as follows. The time a client stays registered at one node is modelled by a normal distribution of mean $\mu_{\mathrm{r}} \in \{10, 30, 60\}$ s and $\sigma_{\mathrm{r}} = 1$ s. For each value, the corresponding distribution for unregistration time is given by $\mu_{\mathrm{u}} \in \{\mu_{\mathrm{r}}, 2\mu_{\mathrm{r}}, 3\mu_{\mathrm{r}}\}$ and $\sigma_{\mathrm{u}} = 1$ s.
4. Since we are trying to characterise the overhead imposed by MODUS during periods of activity, during this tests no appearance or disappearance of nodes is considered.

Tests were carried using as RMS the implementation of HMS, which yields a sending rate of $\tau_{\mathrm{HMS}} \approx 3\ \mathrm{s}^{-1}$ per node, which is practically constant over the whole set of tests.

[6] If the resulting disconnection time was negative, we would consider that no global disconnection takes place.

The experiments were performed on a cluster of four 2.80 GHz Pentium IV Linux PCs, ever which we distributed the N nodes of each simulation. In each case, we measured the number of sent messages per node and from that the instant sending rate, as a function of time, for the various scenarios.

The figures show the measured sending rates in the different experiments, as a function of the number of nodes. For each individual case we also show the rates predicted by the semi–analytical model discussed above. For each experiment we show the prediction dismissing simultaneous unregistration of all clients, and those that evaluate the overhead of expected unregistrations when view changes are respectively minimal and maximal.

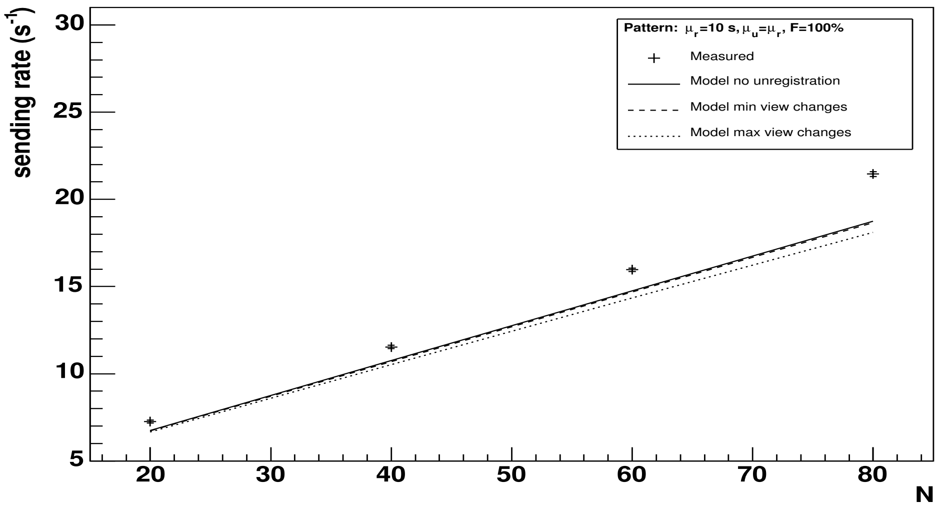


Figure 6.7: Sending rate obtained with $\mu_{\mathrm{r}} = \mu_{\mathrm{u}} = 10$ s and $F = 100$ %.

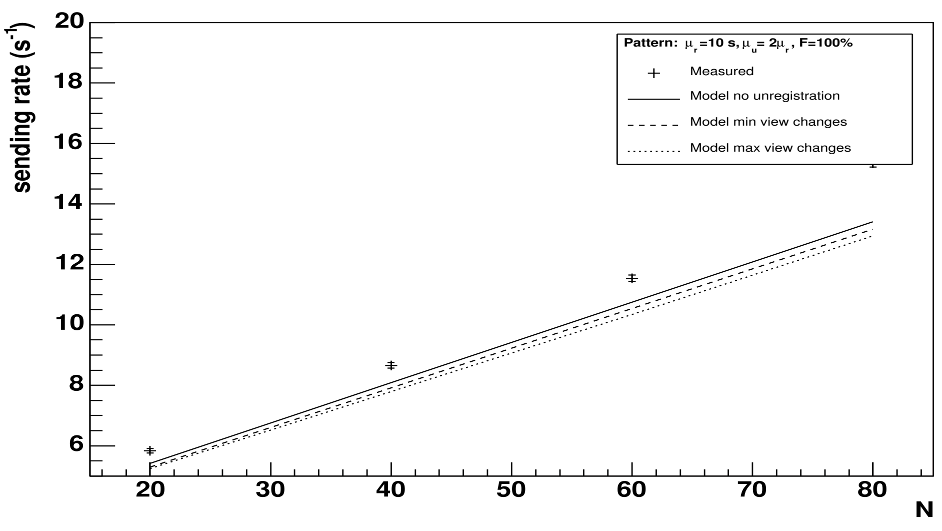


Figure 6.8: Sending rate obtained with $\mu_{\mathrm{r}} = \mu_{\mathrm{u}} = 30$ s and $F = 100$ %.

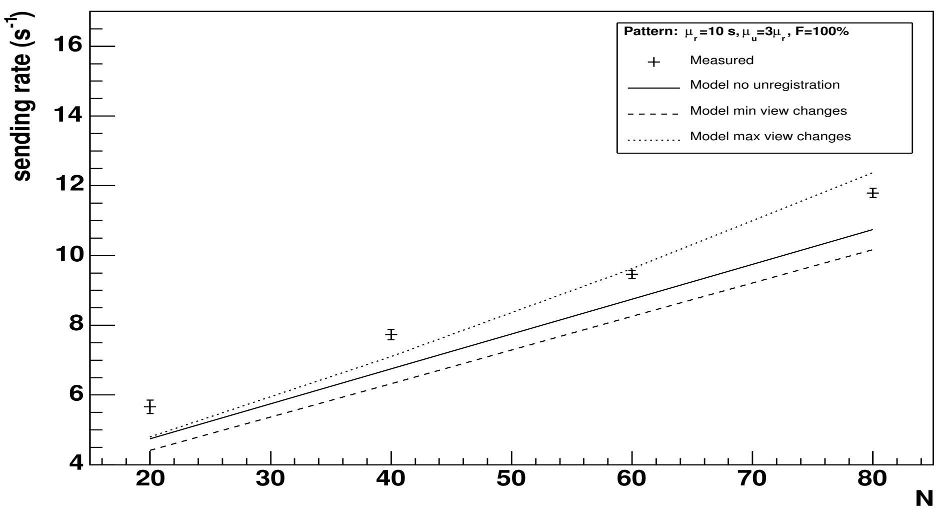


Figure 6.9: Sending rate obtained with $\mu_{\mathrm{r}} = \mu_{\mathrm{u}} = 60$ s and $F = 100$ %.

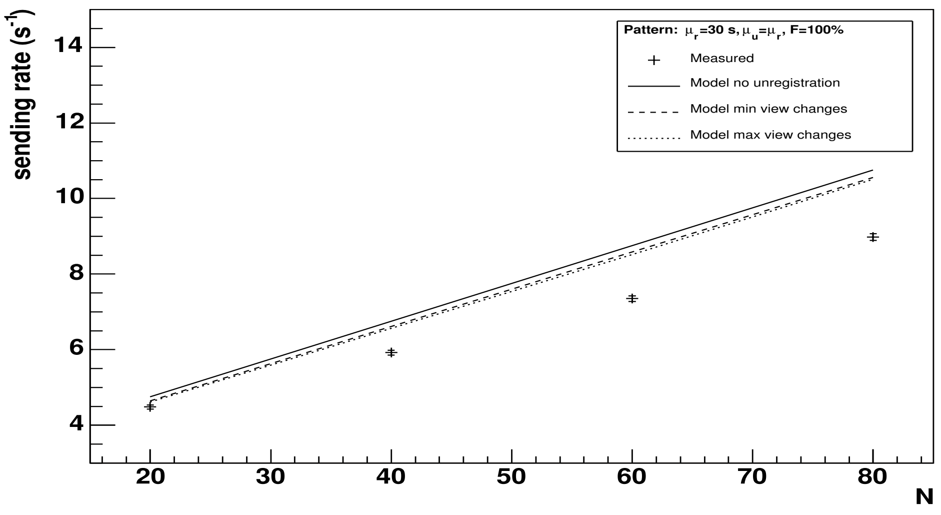


Figure 6.10: Sending rate obtained with $\mu_\mathrm{r} = 10$ s, $\mu_\mathrm{u} = 2\mu_\mathrm{r}$ and $F = 100$ %.

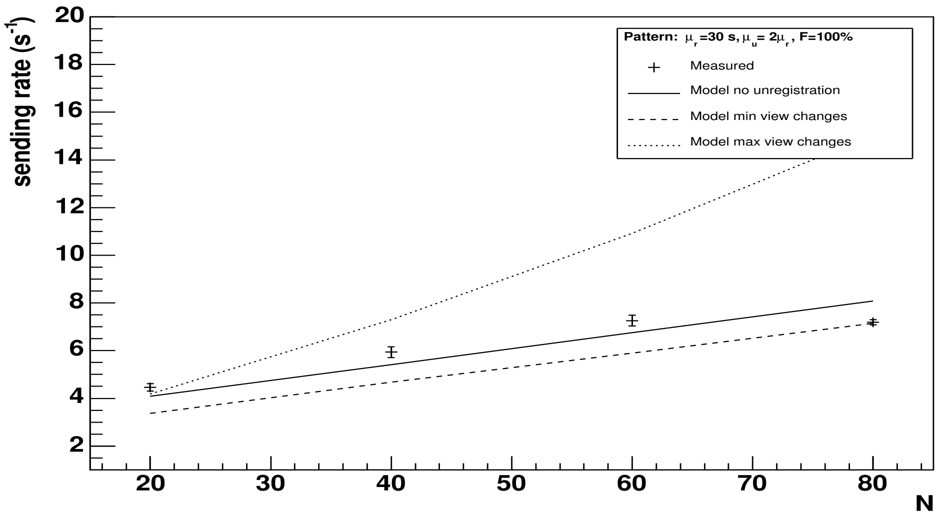


Figure 6.11: Sending rate obtained with $\mu_\mathrm{r} = 30$ s, $\mu_\mathrm{u} = 2\mu_\mathrm{r}$ and $F = 100$ %.

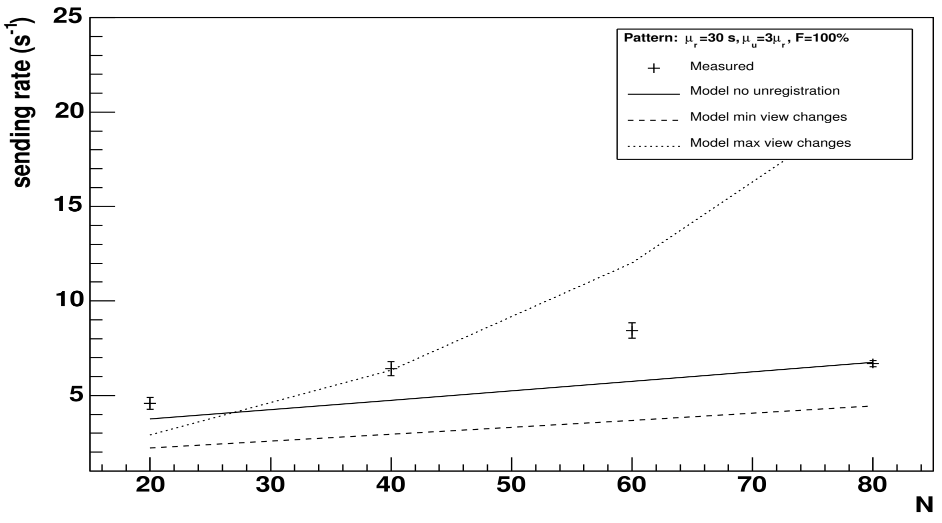


Figure 6.12: Sending rate obtained with $\mu_\mathrm{r} = 30$ s, $\mu_\mathrm{u} = 2\mu_\mathrm{r}$ and $F = 100$ %.

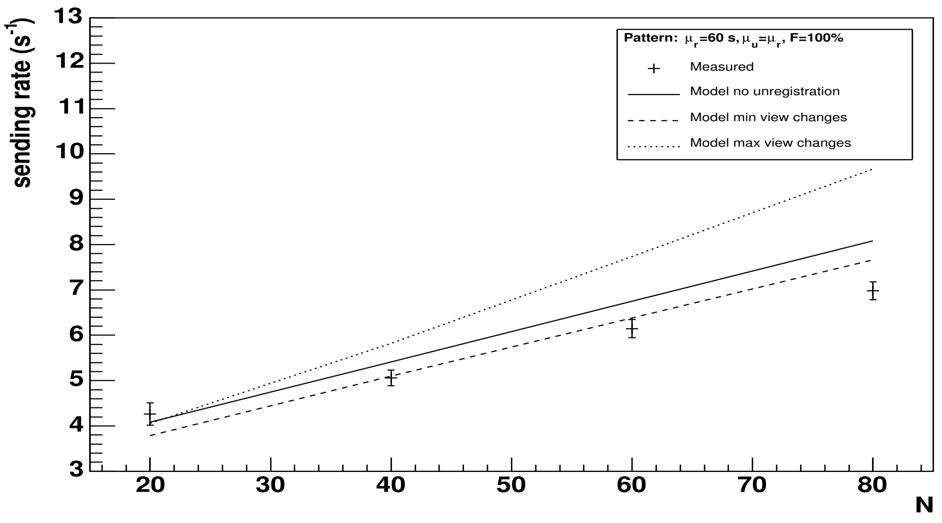


Figure 6.13: Sending rate obtained with $\mu_\mathrm{r} = 10$ s, $\mu_\mathrm{u} = 3\mu_\mathrm{r}$ and $F = 100$ %.

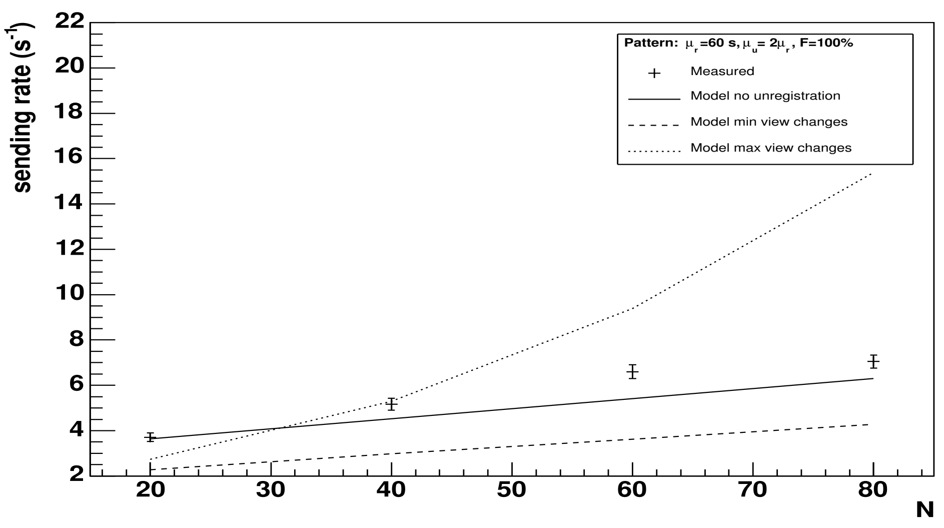


Figure 6.14: Sending rate obtained with $\mu_{\rm r} = 30$ s, $\mu_{\rm u} = 3\mu_{\rm r}$ and $F = 100$ %.

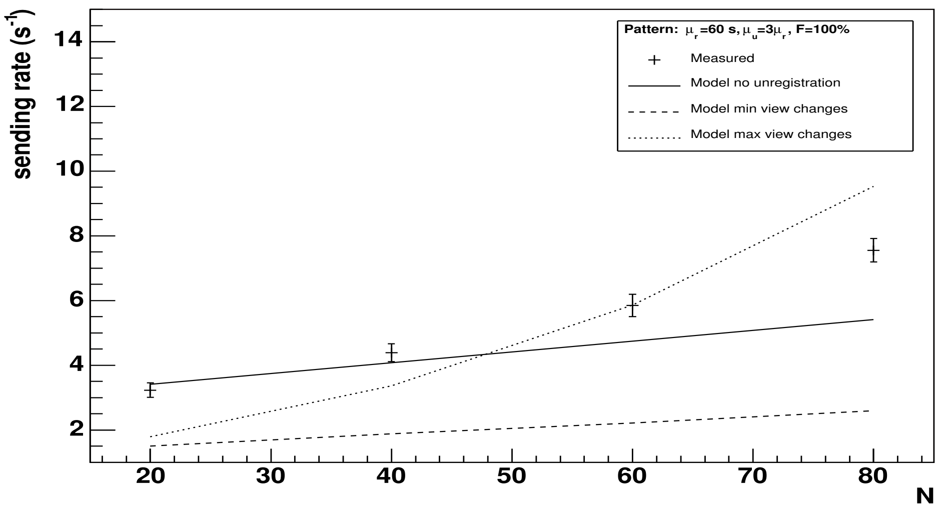


Figure 6.15: Sending rate obtained with $\mu_{\rm r} = 60$ s, $\mu_{\rm u} = 3\mu_{\rm r}$ and $F = 100$ %.

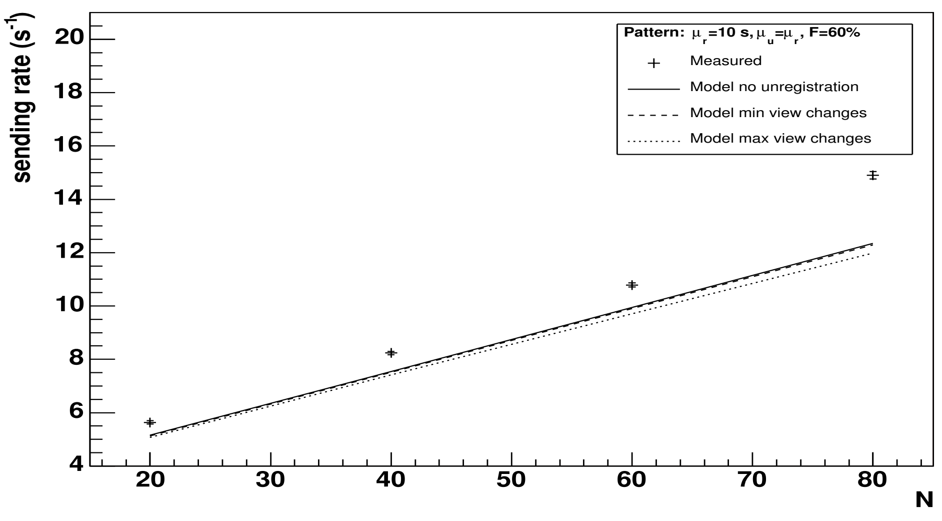


Figure 6.16: Sending rate obtained with $\mu_{\mathrm{r}} = \mu_{\mathrm{u}} = 10$ s and $F = 60$ %.

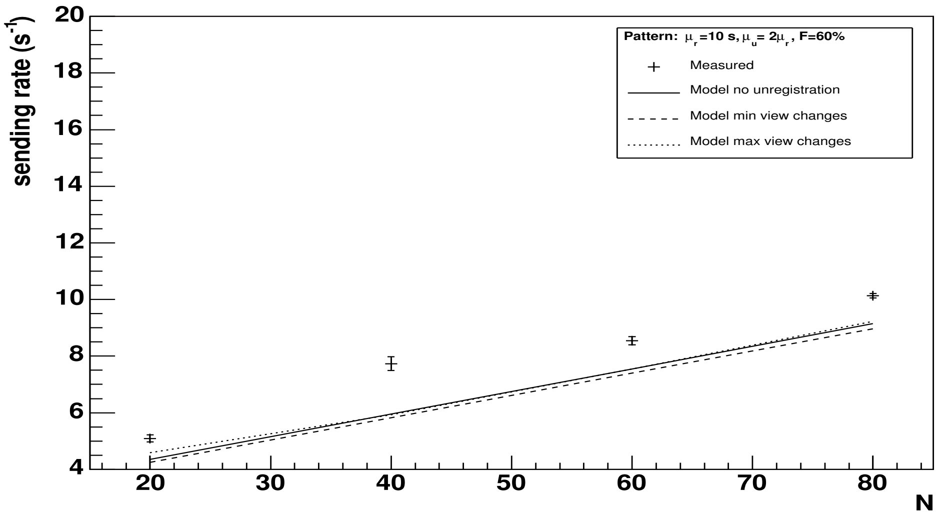


Figure 6.17: Sending rate obtained with $\mu_{\mathrm{r}} = \mu_{\mathrm{u}} = 30$ s and $F = 60$ %.

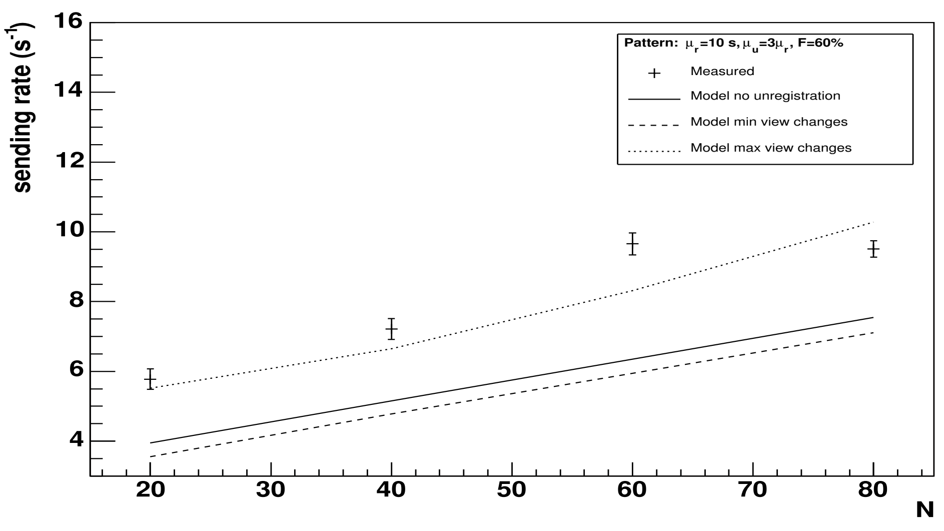


Figure 6.18: Sending rate obtained with $\mu_{\mathrm{r}} = \mu_{\mathrm{u}} = 60$ s and $F = 60$ %.

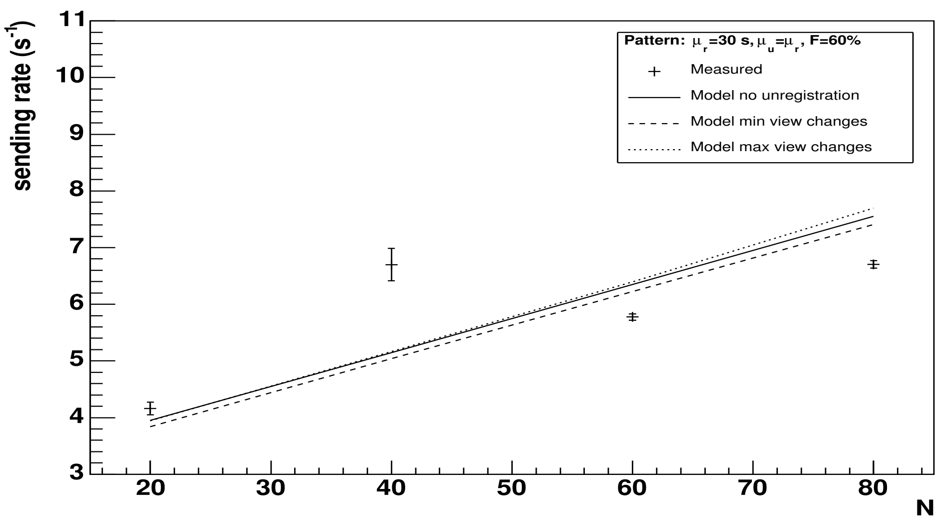


Figure 6.19: Sending rate obtained with $\mu_{\mathrm{r}} = 10$ s, $\mu_{\mathrm{u}} = 2\mu_{\mathrm{r}}$ and $F = 60$ %.

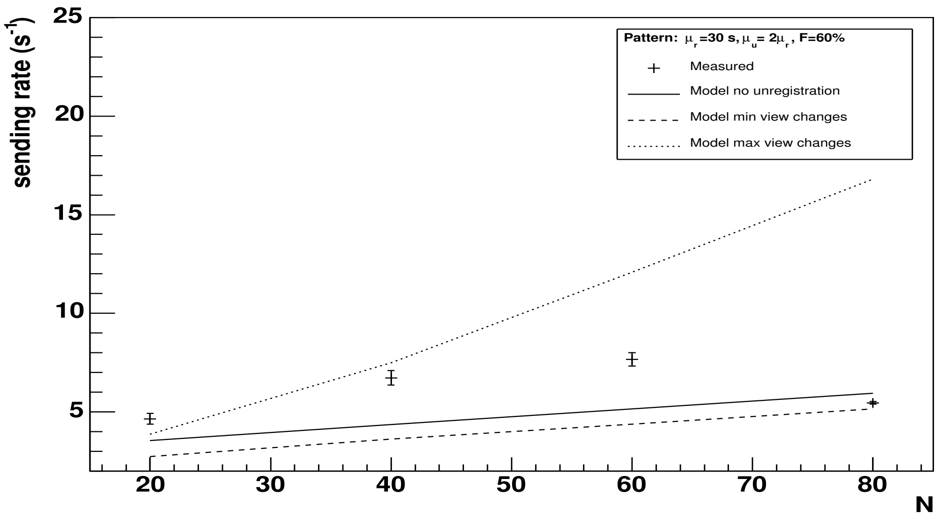


Figure 6.20: Sending rate obtained with $\mu_\mathrm{r} = 30$ s, $\mu_\mathrm{u} = 2\mu_\mathrm{r}$ and $F = 60$ %.

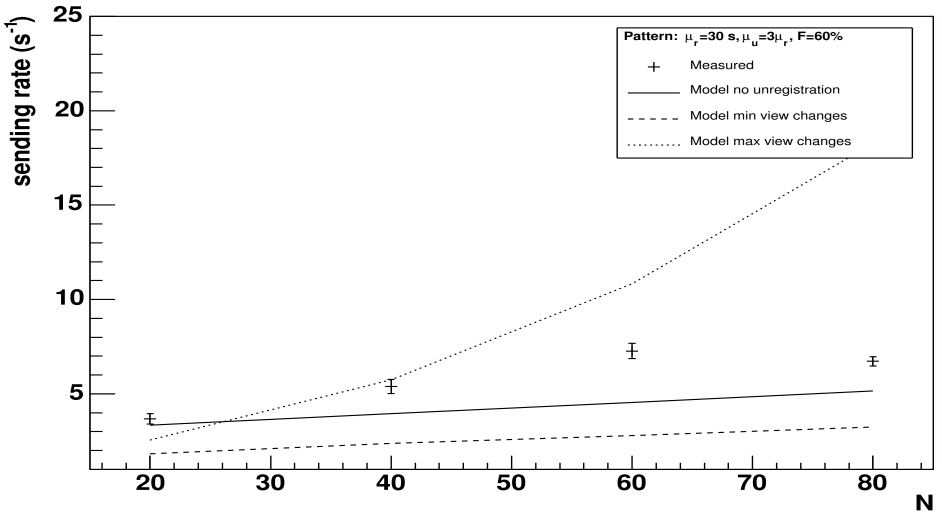


Figure 6.21: Sending rate obtained with $\mu_\mathrm{r} = 60$ s, $\mu_\mathrm{u} = 2\mu_\mathrm{r}$ and $F = 60$ %.

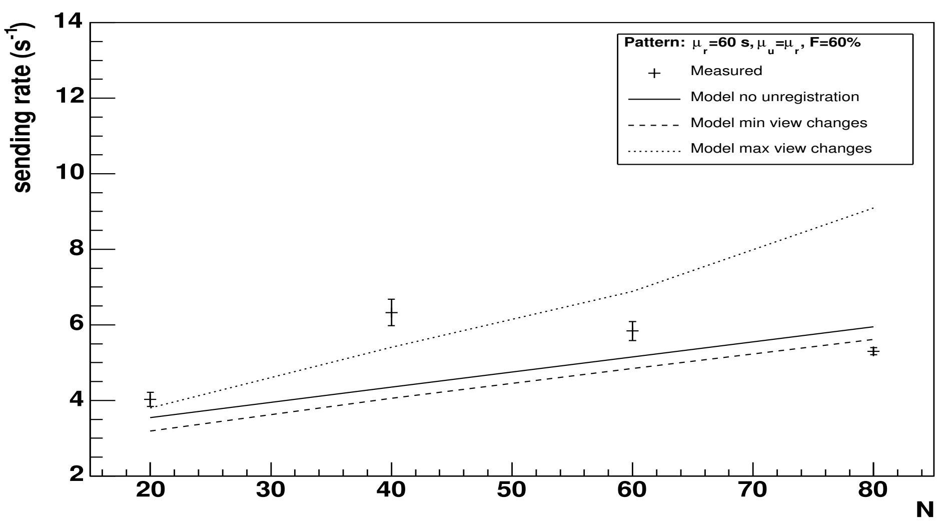


Figure 6.22: Sending rate obtained with $\mu_\mathrm{r} = 10$ s, $\mu_\mathrm{u} = 3\mu_\mathrm{r}$ and $F = 60$ %.

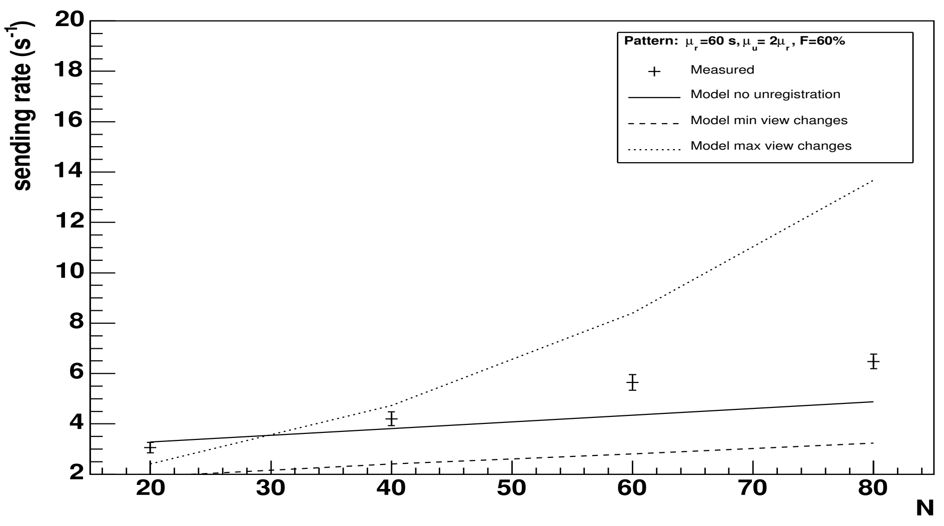


Figure 6.23: Sending rate obtained with $\mu_\mathrm{r} = 30$ s, $\mu_\mathrm{u} = 3\mu_\mathrm{r}$ and $F = 60$ %.

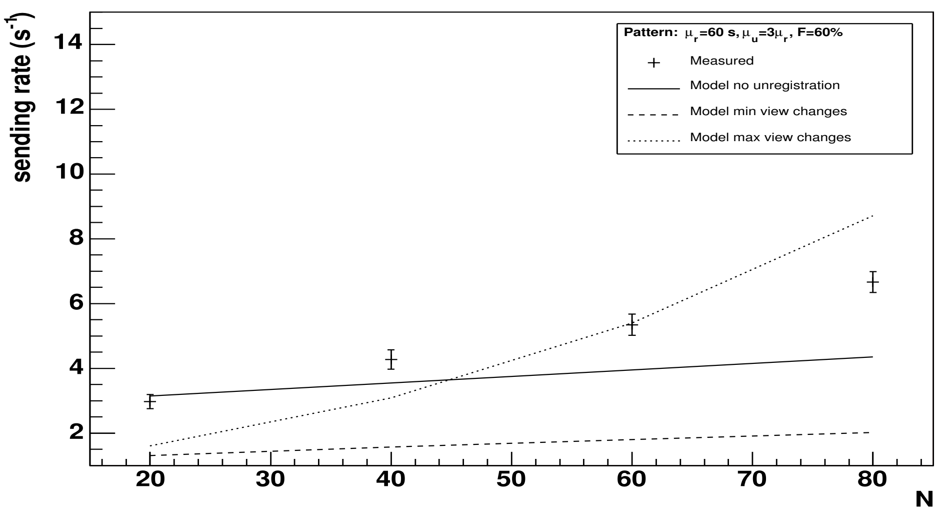


Figure 6.24: Sending rate obtained with $\mu_{\mathrm{r}} = 60$ s, $\mu_{\mathrm{u}} = 3\mu_{\mathrm{r}}$ and $F = 60$ %.

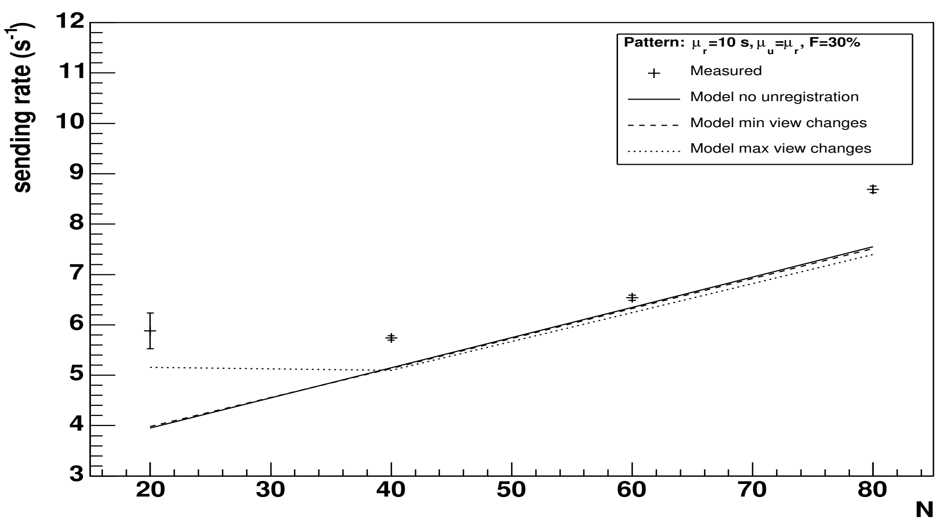


Figure 6.25: Sending rate obtained with $\mu_{\mathrm{r}} = \mu_{\mathrm{u}} = 10$ s and $F = 30$ %.

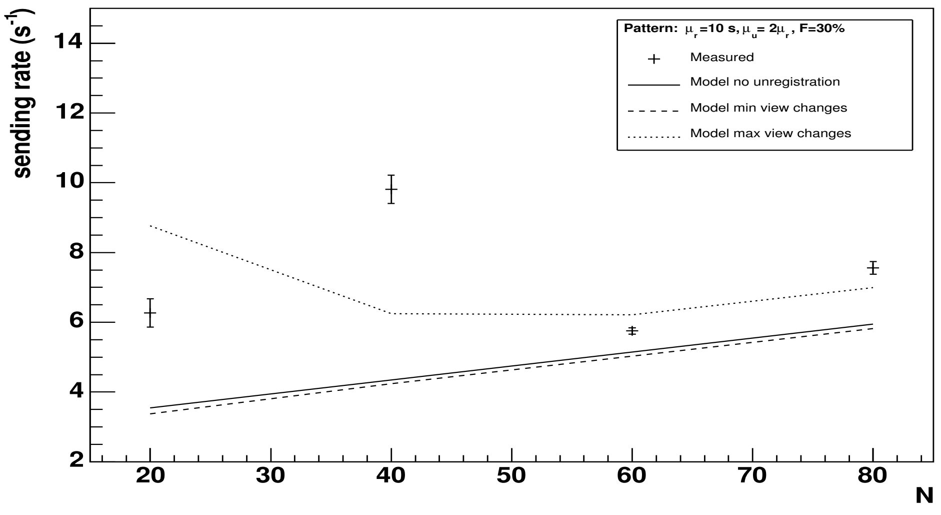


Figure 6.26: Sending rate obtained with $\mu_{\mathrm{r}} = \mu_{\mathrm{u}} = 30$ s and $F = 30$ %.

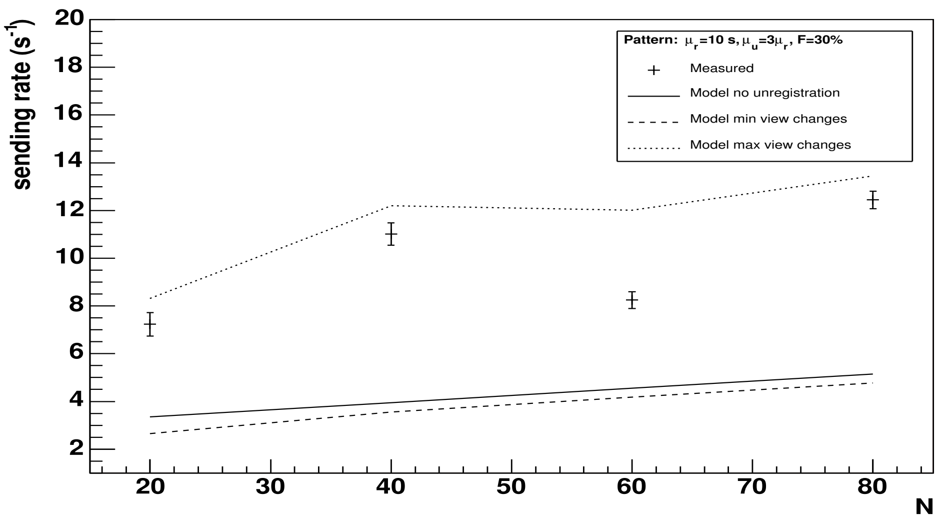


Figure 6.27: Sending rate obtained with $\mu_{\rm r} = \mu_{\rm u} = 60$ s and $F = 30$ %.

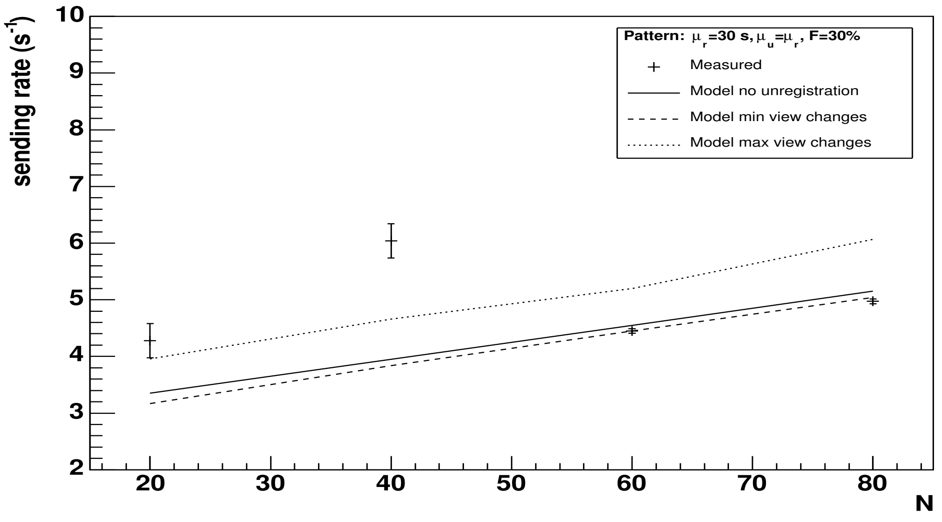


Figure 6.28: Sending rate obtained with $\mu_{\rm r} = 10$ s, $\mu_{\rm u} = 2\mu_{\rm r}$ and $F = 30$ %.

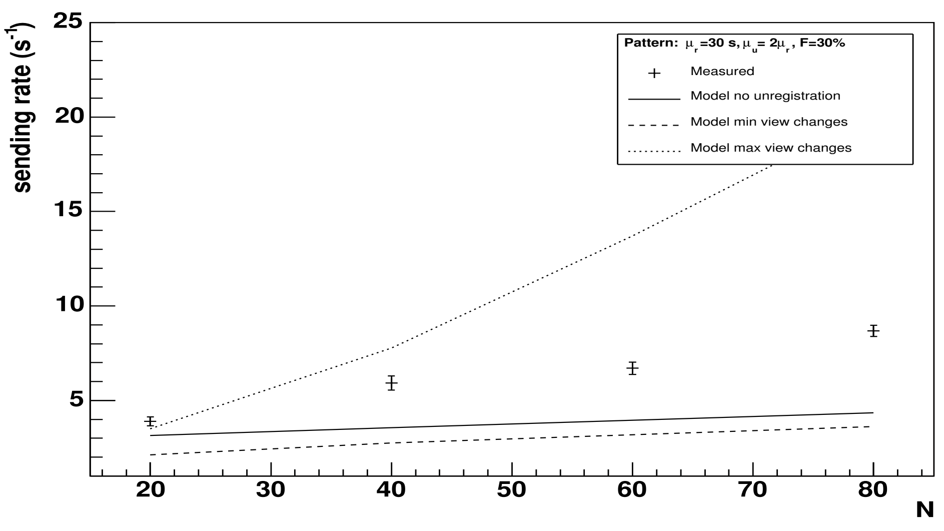


Figure 6.29: Sending rate obtained with $\mu_{\mathrm{r}} = 30$ s, $\mu_{\mathrm{u}} = 2\mu_{\mathrm{r}}$ and $F = 30$ %.

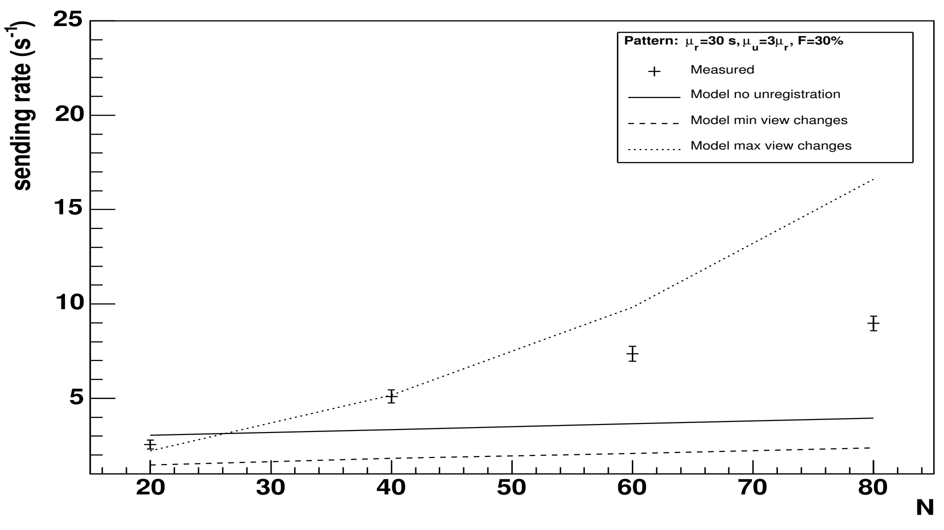


Figure 6.30: Sending rate obtained with $\mu_{\mathrm{r}} = 60$ s, $\mu_{\mathrm{u}} = 2\mu_{\mathrm{r}}$ and $F = 30$ %.

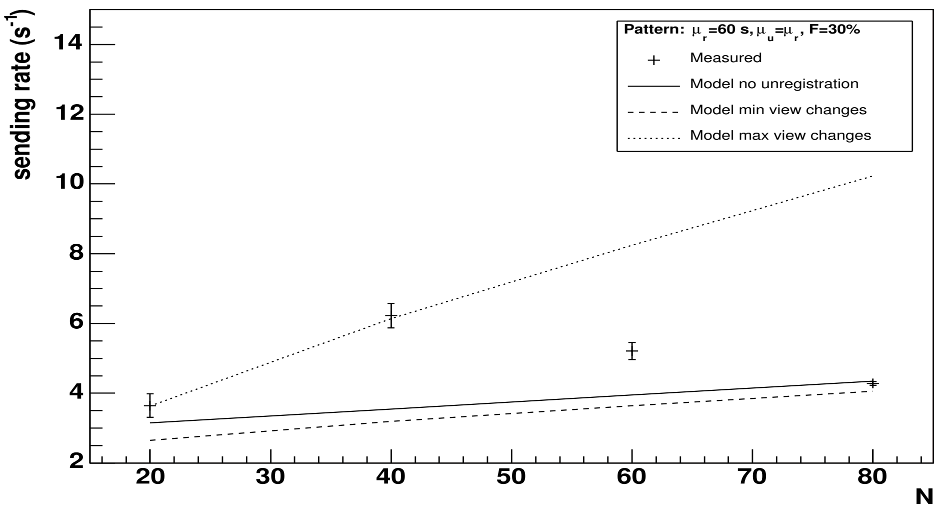


Figure 6.31: Sending rate obtained with $\mu_\mathrm{r} = 10$ s, $\mu_\mathrm{u} = 3\mu_\mathrm{r}$ and $F = 30$ %.

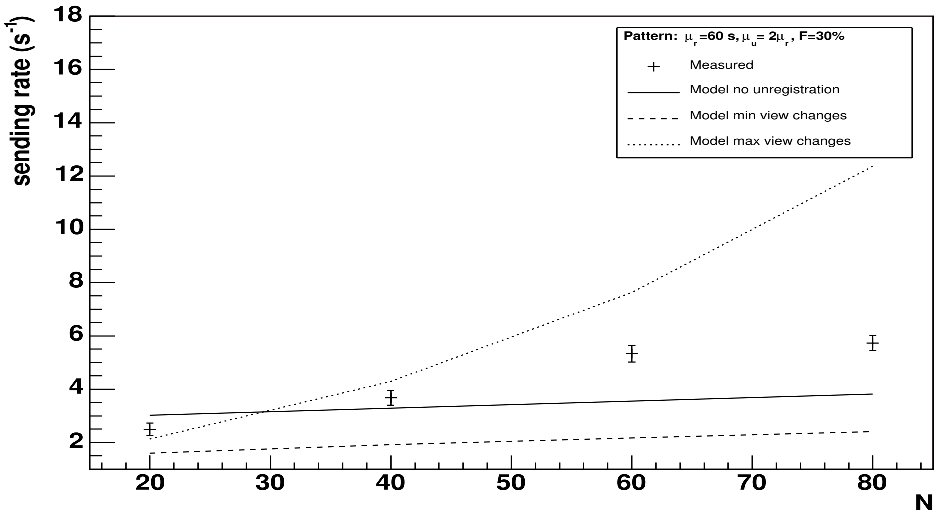


Figure 6.32: Sending rate obtained with $\mu_\mathrm{r} = 30$ s, $\mu_\mathrm{u} = 3\mu_\mathrm{r}$ and $F = 30$ %.

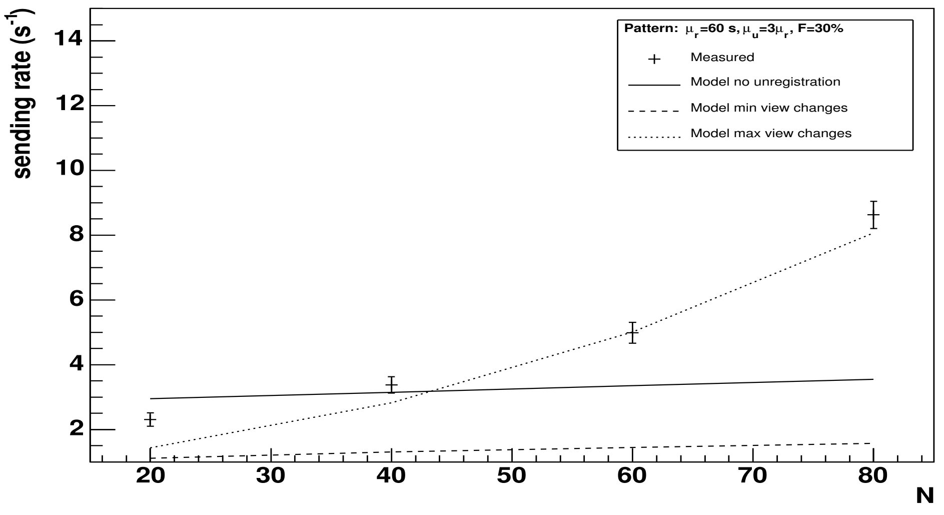


Figure 6.33: Sending rate obtained with $\mu_\mathrm{r} = 60$ s, $\mu_\mathrm{u} = 3\mu_\mathrm{r}$ and $F = 30$ %.

The first nine plots, figs. 6.7-6.33, correspond to the experiments with the largest proportion of connection time, $\mu_{\mathrm{u}} = \mu_{\mathrm{r}}$, and to situations in which all nodes in the system hold membership clients. The plots show how, as the frequency of individual registrations decreases, the potential contribution of the unregistration and reconfiguration overhead (whose boundaries are shown as dashed lines) becomes more important, so that the obtained rates deviate from the linear behaviour predicted for continuous operation. On the other hand, if such frequency is too high, other factors may appear that are not included in the model, such as repeated sending of messages as they may be lost due to flow peaks, which may cause the continuous rate to be non linear.

If the percentage of active nodes decreases to only 60 % of the system nodes (see figs. 6.16-6.24), the occurrence of overlapped unregistration of all clients becomes more likely, and the results deviate from the continuum prediction, but are still far from the maximum rates predicted by the most pessimistic reconfiguration pattern, except for the longest mean unregistration times, for $\mu_{\mathrm{r}} = 60$ s (see figs. 6.22, 6.23, 6.24).

In figs. 6.25-6.33, analogous experiments with only $F = 30$ % of active nodes are depicted. In these cases, the overhead of full unregistration becomes more important, and the experimental results get close to the worst prediction for $\mu_{\mathrm{r}} = 60$ s.

Generally speaking, MODUS represents an overhead with respect to the message consumption imposed by a RMS that is running continuously. During periods in which membership services are not required, however, such overhead my be compensated by the sustained consumption of RMS. The critical fraction of time γ_{act} under which MODUS turns out to be the most efficient option, depends on the particular RMS considered. The following plots show the results obtained for $1/\gamma_{\mathrm{act}}$ in our experiments.

From the previous experimental results we may predict a range of operation regimes for which an application will benefit from the use of MODUS instead of a given conventional RMS, such as HMS. In particular, we observe that the overhead of MODUS will in most cases be compensated if membership services are only required for less that 30 % of the time.

A reasonable alternative to the execution of RMS on all nodes, which might be more efficient than MODUS from the point of bandwidth consumption could be running RMS alone, but enabling every node to switch off its individual membership service when no local client exists for membership information. With nodes joining and leaving the cluster as their local applications require and release membership services, the RMS would be forced to install new views with every change, thus generating a number of message broadcasts. The number of changes RMS would undergo would then be determined by the shortest from t_{view} and the mean time between registration/unregistration events. If changes were faster than view installation, the number of views in a total period T would be determined by their average duration, $n_{\mathrm{views}} = T/t_{\mathrm{view}}$. Otherwise, the number of installed views would be given by the number of registrations and unregistrations, $n_{\mathrm{views}} = 2T/(\mu_{\mathrm{r}} + \mu_{\mathrm{u}})$. Since only nodes with active clients would be

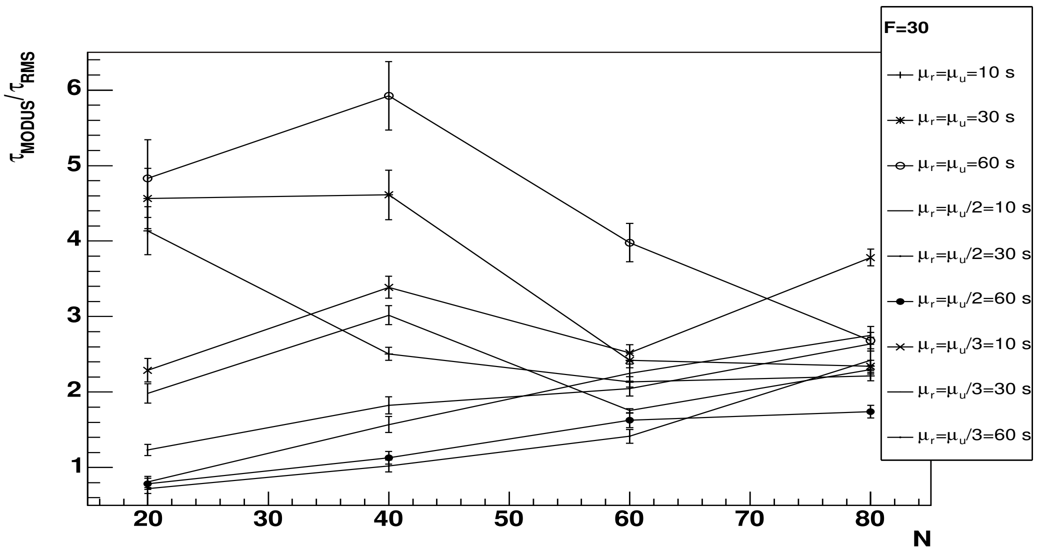


Figure 6.34: Ratio between the sending rate of MODUS and that of RMS running continuously on all nodes, for $F = 30$ % of active nodes.

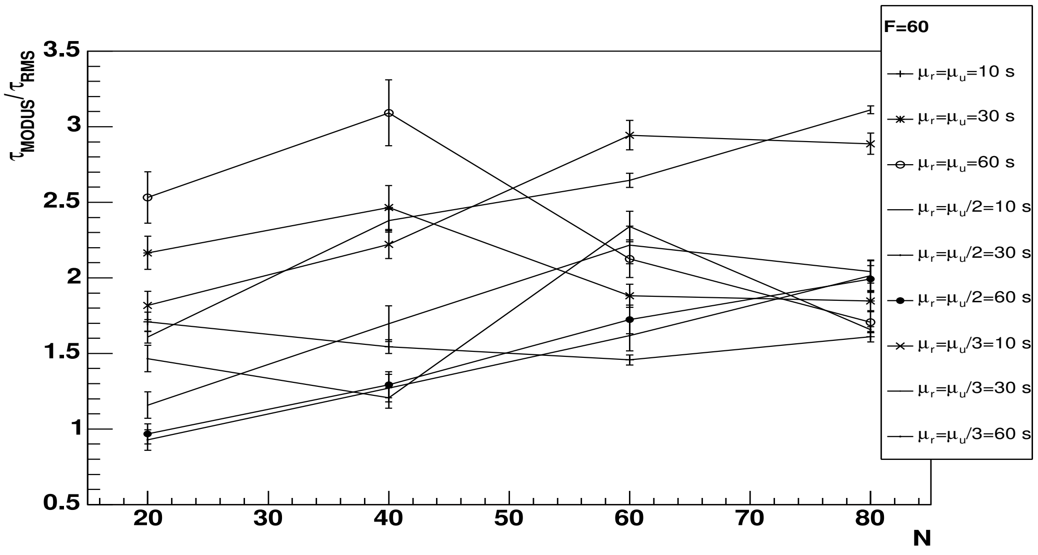


Figure 6.35: Ratio between the sending rate of MODUS and that of RMS running continuously on all nodes, for $F = 60$ % of active nodes.

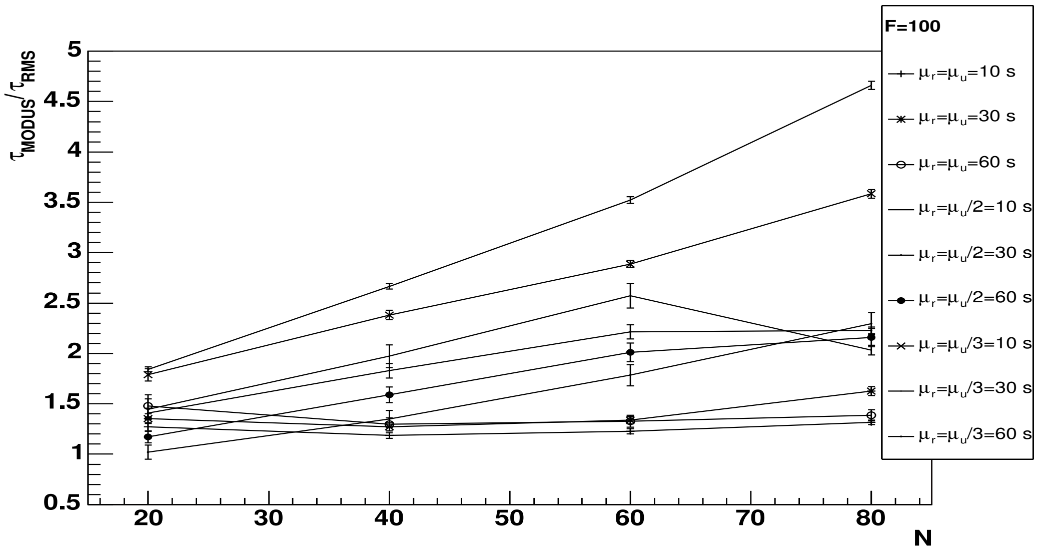


Figure 6.36: Ratio between the sending rate of MODUS and that of RMS running continuously on all nodes, for $F = 100$ % of active nodes.

participating of the membership protocol, the group views formed in such way would contain in average $F\ N\ P_{\text{reg}}$ members, which may be not enough for the requirements of the user applications. For instance, a group key agreement protocol might be requiring a minimum group size in order to proceed.

The higher flexibility of MODUS with respect to RMS run with such configuration lies in the possibility of a distributed switching off without damaging the service offered to user applications. The size of the group in the case of MODUS will be determined by the fraction of available nodes, if message losses are neglected. Thus, in our experiments, the average size will approach the maximum N, i.e. it will be $\frac{100}{FP_{\text{reg}}}$ times larger than the average RMS group, except for periods of global unregistration. This is illustrated by figs. 6.37, 6.38, which show the extreme cases with the maximum and minimum frequency of registration–unregistration for the maximum and minimum fraction of active nodes, respectively.

From the results above we conclude that MODUS will be an advantageous alternative to a conventional RMS when the applications require membership information only for finite periods, but can do without this service for the most of their running time, and when their operation has also requirements on the size of the group.

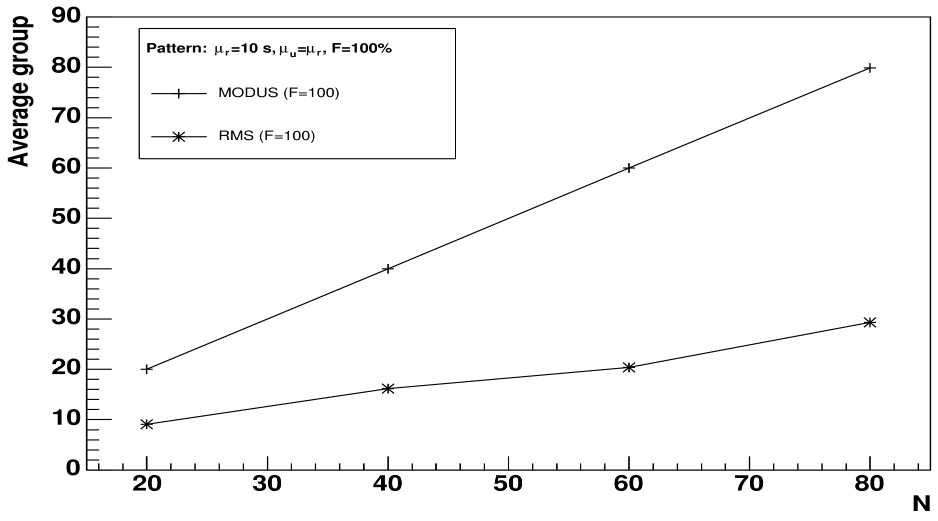


Figure 6.37: Comparison of the average group size with MODUS and RMS alone, when the latter is run only during local activity times, for $\mu_{\mathrm{r}} = \mu_{\mathrm{u}} = 10$ s and $F = 100$ % of active nodes.

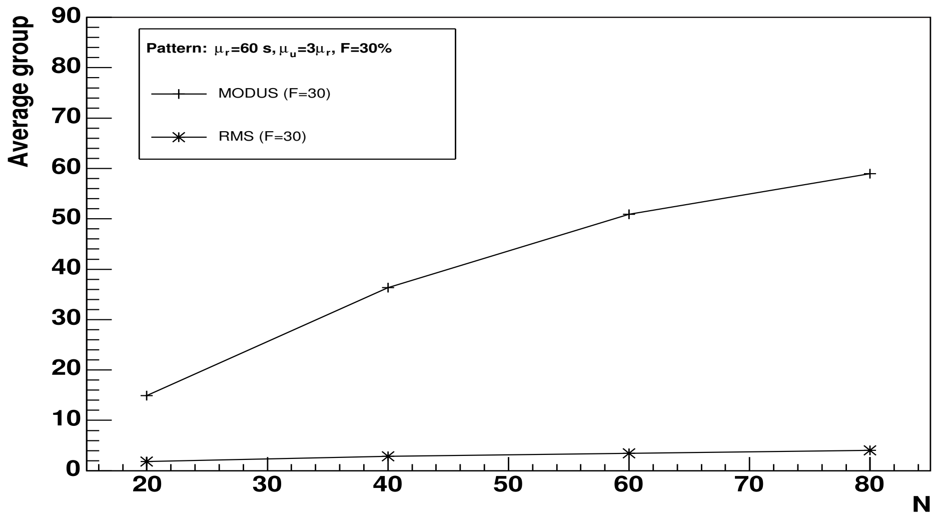


Figure 6.38: Comparison of the average group size with MODUS and RMS alone, when the latter is run only during local activity times, for $\mu_{\mathrm{r}} = \mu_{\mathrm{u}}/3 = 60$ s and $F = 30$ % of active nodes.

6.5 A Practical Case of Use

In this section we describe an example application that could benefit from the use of membership services in an On–Demand fashion.

Let us picture a meeting among a relatively high number of participants (up to a hundred) that join a conference session and are supposed to carry some collaborative work. As the example application, we may consider a distributed agenda with secure key exchange. Each participant is equipped with either a laptop or a PDA, all of them connected to the wireless network and running the agenda. The application should be completely distributed, and thus any participant is entitled to read and make changes to the schedule.

A typical session would begin by establishing a shared private key for the group to enable efficient communications security. This will require consensus among at least a subset of the initial participants, and thus the application will demand the membership agreement for the duration of this phase. Depending on the application, a minimal number of participants can be required in order to establish a key, so that if no such group can be formed, the application can not proceed. After a key has been established, the participants will mostly consult the agenda in a read–only manner. The distribution of the secure key to new members does not necessarily need the participation of all the group, but can be done in a one-to-one basis. The duration of the agreement requirement is then finite, and the membership services can be turned off.

Whenever a member decides to write to the agenda, changes must arrive to all participants with total order guarantees, to avoid inconsistencies caused by concurrent write operations. Therefore write operations should be performed within the consensus provided by membership guarantees.

Some of the participants in the meeting may have their laptops connected to the AC supply, and thus be more willing to play the most energy–consuming roles in the application. It is important to notice that this decision should not be taken *a priori*, as would be the case in a (partially) wired network, where some nodes might play the role of servers while the others act as clients. On the contrary, the application should respect the symmetry among all participants inherent to ad hoc networks, so that any node can disconnect or reduce its participation due to power shortage without affecting the overall correct operation of the application. In the case of our agenda, some plugged-on participants could invoke `MODUS.joinGroup` at the beginning of the session and stay available as membership participants, whereas the lightest participants should only make themselves available when they are interested in agreement. The application may use the membership information it receives to enforce its own guarantees, e.g. the minimum number of participants required to establish a secure key or to allow a certain change to the data.

Other applications, as distributed file sharing, could exploit On–Demand membership semantics to achieve their particular goal. Leader election is also trivial taking advantage of MODUS semantics.

These and other distributed applications could obviously be implemented without the support of membership semantics. However, that would imply much more complicated and failure prone implementations, as the effort of achieving and maintaining the required agreement would fall on the applications developer. That cost would be replicated if various components exist with the need for membership information.

Chapter 7

Membership Estimation for Dynamic Networks

7.1 Specific Membership Service for Ad Hoc Networks

The MODUS protocol, described in the previous chapter, is a useful alternative in order to provide the On–Demand semantics when a conventional membership service is available. As shown in the previous section, the On–Demand approach alone already allows significant saving in bandwidth and energy consumption. Nevertheless, in particular scenarios it may be more convenient the usage of specific protocols, better adapted to the characteristics of each environment.

In the case of ad hoc networks the special features of the system can render impractical the usage of a conventional protocol as the RMS required by MODUS. For instance, if HMS is to be used, an initial knowledge is required about the identities of all possible members of the system. In an ad hoc network such information may not be available, since members are not in general preconfigured and the topology is a priori unknown. The same limitation applies to conventional membership services which require only a partial knowledge of possible members to start up, or in general to any protocol that needs some information about the composition of the system.

On the other hand, the absence of fixed hosts advises against the use of centralised protocols. Even the usage of a master, as in the case of HMS, which leads changes through several views during its life will affect negatively the performance of the protocol, since the extra work that has to be done by the master node will probably impose a higher energy consumption on this particular node, and thus make it more prone to failures.

For all these reasons it is convenient to deploy a specific protocol that better conforms to such distinguishing features of an ad hoc network. In particular, no a priori knowledge of the node identities should be assumed, and maximally decentralised protocols should be chosen.

7.1.1 An Alternative Membership Architecture

A suitable alternative to deploy a membership service that provides the On–Demand service is to base it on a service that provides an estimation of the current membership, without strict guarantees, on top of which agreement rounds are launched according to the On–Demand semantics, as in [7] the conventional membership service is achieved with an estimation phase followed by a consensus run. Thus we propose a two-folded membership service (see Fig. 7.1) where both components offer to higher level applications the most general interface of a membership monitor 3.1.

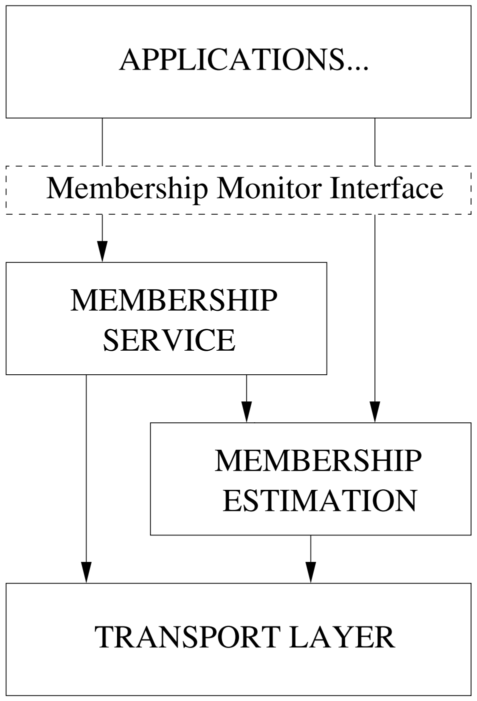


Figure 7.1: Architecture of a specific On-Demand Membership Service.

The lowest component provides only an estimation of the group composition. This supplies the initial information that in a conventional system may be a priori available about the network addresses of the members. The estimation component tends to keep each node's information up to date, but since it does not provide consistency guarantees, a stronger protocol is required when the conventional membership semantics is to be provided. The upper membership component will take charge of this task, providing its client applications with the conventional guarantees of a membership service [22]. Some of its distinguishing features are the following.

- The strict component does not run continuously, but implements the On–Demand semantics discussed in the previous sections.
- The design of the membership service should not rely on the existence of a master that starts the agreement rounds, since the rate of failures is high and that would force extra (maybe costly) recovery protocols to be run often, as the master changes. It is thus desirable that the proposed service is completely distributed, and that the protocol can be simultaneously launched by different nodes.
- Each run cannot rely on the results of a previously reached agreement, since operation is not continuous and the latest confirmed view does not need to be valid any longer. Thus, each time an agreement is demanded, the protocol will proceed in identical way, retrieving an initial set from the estimated membership component and trying to reach an agreement about that composition.

The interaction between both components must respect the inherent system limitations. In particular, the requirements on the liveness of the protocols should be relaxed as far as this does not break the desired consistency properties. These considerations lead to an approach where addition of new nodes to the confirmed group can be delayed, if detected while an agreement is being forced. Those new nodes will not take part of the current agreement, but will only participate in further rounds, if required. Conversely, the failure of a node while the membership service is running must be promptly notified, as a missing member is sensitive information to any application that is listening to membership changes (e.g. if it is trying to complete a transaction within the confirmed group).

7.2 Membership Estimation Service

The estimation component was introduced in the former section as a basis for the operation of a strong membership service. Nevertheless, it is a more general service whose information will be useful for other distributed components. For instance, a consensus algorithm could need a set of processors to be used as the agreement set.

In other words, the membership estimation service provides an approximate information about the group composition that can be useful to any other component which does not require strong consistency (see [60]). Therefore, it can be made directly available to higher level applications, as it offers the public membership interface.

The rest of this chapter is devoted to a study of an efficient approach to provide this estimation service in an ad hoc network. Different to existing studies about the optimisation of protocols for ad hoc networks, our analysis takes into account the necessary presence of other services, in particular routing. Given the particular characteristics of this kind of network, the energy consumption is to be minimised by any protocol. However, this clashes with the aim to provide as live as possible information about the group composition. Moreover, the routing and estimation services do not need to be isolated, but can be combined in a single component.

Thus, the fact that both services have to be simultaneously provided affects the architectural decisions, as well as the energy saving policies. Here we compare two possible architectures that provide both services and analyse their performances in terms of power consumption and quality of the membership estimation [81].

7.3 Interaction of Routing and Membership Estimation

Several approaches may be chosen for the estimation component. Its operation may be based on gossip, so that when a particular node detects an addition or an announced disconnection, such information is spread across the group. There is however an interesting consideration to be made here. Our system model (see sect. 6.2) is assuming the existence of an underlying routing service which enables all our protocols to address messages to individual nodes in the system. Such routing protocol will maintain at least partial information about the composition of the system, and will also impose a certain energy consumption. Therefore it involves two fundamental aspects the estimation service is concerned about.

Moreover, there exist routing protocols that keep the global composition of the system at every node, e.g. OLSR, and can provide membership information by themselves. It is then reasonable to perform a joint study of the energetic efficiency and the quality of the membership estimation of a gossip–based membership estimation and a routing service, and compare the results to the estimation obtained by using only a routing service that holds full information about the system.

7.3.1 Routing Protocols

The multiple existing proposals for routing protocols in ad hoc networks can be classified in two groups, namely reactive (also called "on-demand") and proactive.

Reactive algorithms, as DSR [82], AODV [83] and TORA [84], only operate when a new packet is to be sent and no calculated valid route exists from the sender to the receiver. The route search process introduces some delay in the delivery, but the overload due to network maintenance messages is very low. On the contrary proactive algorithms, as OLSR [85] and DSDV [86], periodically exchange data about routes and system state in order to refresh the system topology. They imply a constant energy consumption by all nodes, and introduce a higher overload in the network.

A number of studies have been carried in order to characterise the performance of both approaches with respect to different parameters and in various working scenarios [77, 87, 88, 89].

Here we focus on the joint study of the energy consumption and the quality of the provided global information about the group membership. For our study we have selected two routing protocols, representative of each type and for which stable implementations exist for different platforms, namely DSR, as reactive protocol, and OLSR, as proactive protocol which maintains global information about the system composition at every node.

OLSR OLSR (Optimised Link State Routing Protocol) [90, 85] is a proactive routing algorithm built as an optimisation of link state algorithms by reducing flooding overload only to a set of nodes called MPR (Multipoint Relays). To our knowledge, it is also the only one to maintain global information on the composition of the system at every node. Its proactive nature ensures that routes are available at any moment.

From an operational point of view, OLSR handles two types of messages.

- *Hello* messages are used to discover links and to test their unidirectional or bidirectional character. They are exchanged periodically between neighbours to probe the state and directionality of links and to verify the responsiveness of nodes.

- *TC* messages are used to build the overall network topology. Each node selects its set of MPR nodes from the senders of received *Hello* messages and periodically broadcasts a *TC* message to the entire network with part of the received information. This message is only forwarded by MPR nodes and it is used by every node to update the global composition of the system.

The information received by a given node through *Hello* and *TC* messages is kept for a certain period of time. If it is not refreshed by new messages, however, the corresponding links will be considered to have failed and will be removed from the locally maintained image of the system composition.

The behaviour of OLSR is determined by a number of parameters, among which the most relevant for our comparison are the following.

- H_i represents the time interval between consecutive *Hello* messages;
- TC_i is the corresponding interval between TC messages;
- H_t is the time of validity of the last *Hello* message (typically $H_t \propto H_i$); and
- TC_t, the time of validity of the last TC message (with $TC_t \propto TC_i$).

The last two parameters determine when the information held by one node is considered obsolete and discarded, thus they determine the liveness of the protocol regarding the detection of failed nodes or links.

DSR DSR (Dynamic Source Routing) [82] is one of the most widely used reactive algorithms. Its reactive nature implies that every node maintains only those routes that have been previously demanded (or parts of such routes, if they have taken part on their calculation) and have not yet been discarded due to broken links.

In a very condensed way, the operation of DSR is as follows. When a packet is to be sent to an individual node, and if no pre-calculated route to the receiver exists, the sender produces and broadcasts a *Route Request* message, containing its identity and that of the destination, a unique message identifier and a path list initially holding the sender alone. The neighbouring nodes that receive the *Request* add their identifier to the route path list and broadcast the modified message. After receiving it, the target replies to the source with a *Route Reply* message that contains the whole path followed by the message from the sender to the destination.

Since *Route Requests* are only issued when there are messages to be sent, no constant overhead is imposed by DSR, at the price of a certain delay when delivering a message to a node for which no route is yet known. On the other hand, as a node N_i only holds information on nodes for which a route including N_i exists, there is no guarantee that the global composition of the system is locally available.

Liveness is ensured by requiring the confirmation of data packets reception at each hop. If a node misses such confirmation from the next node, it can retry the sending for a number of times. After a certain number of failed retransmissions, however, it generates a *Route Error* message to notify the original source about a broken link in the route.

7.3.2 Epidemic Style Membership Estimation

To our knowledge no reactive routing protocol provides every node with information about the global composition of the system. Thus, if such approach is used for routing, an extra service should be implemented to perform the membership estimation. A good strategy to achieve this in a wireless ad hoc network is to build a gossip–style protocol, that will indeed impose some extra energy cost, but which can be tuned independently of routing requirements.

Epidemic or gossip protocols were first introduced in [91] and are widely used for spreading information in large distributed systems [92] without strict guarantees but high probability of success. Their use is typical for providing group communication services in wireless networks [48, 93, 50, 94, 95], and has also been proposed for achieving computational tasks in large groups [96].

As a membership estimation service, its use was first proposed in [41], whereas [97] defined a gossip–like failure detection system in charge of monitoring changes to the network in a wired environment. Scalable versions of the latter made use of the hierarchy of the network.

Epidemic algorithms are characterised by their robustness against process or link failures and their easy implementation [92]. The basic operation of a general epidemic protocol is as follows. Each process in the distributed system forwards the received information to a randomly chosen subset of peers. The dynamics mimics the spreading of an infectious disease. The parameters that govern these protocols, i.e. the quantity of information every node can maintain, the number of times the same message can be forwarded and the number of *infected* peers at each step allow a wide adaptivity and the existence of range of variated gossip algorithms.

The use of gossip protocols for routing in ad hoc networks was analysed in [98]. In that work several gossip variants are proposed and studied for ad hoc networks. We use one of them to adapt the classic gossip–style failure detection in [97], so that instead of selecting a random destination for the gossipped message, each receiver probabilistically decides whether to forward data. This yields a gossip–like protocol which provides each node with an estimation of the changes to the composition of the global network.

In particular, the gossip protocol we use maintains at every node a table of connected nodes, *mbship_est*. Each entry is of the form

$$(N,\ state,\ hb,\ T_{\rm hb})\,,$$

where N is the node's identity; the second field holds the state of the node within the group, as $state \in \{CONNECTED,\ FAILED\}$; hb is an integer value, or heartbeat; and $T_{\rm hb}$ is the time at which the value hb was last updated.

The protocol makes use of a single type of messages, namely **Gossip(*table*)**, containing as argument the local *mbship_est* table of the sending node. The basic operation is described in this section, and the schematic pseudocode can be found in fig. 7.2, where the protocol has been divided in two states or phases, namely STARTUP and NORMAL.

At startup, the single entry contained in a node's table is the local one, with

$hb(localNode) = 0$ and $T_{\mathrm{hb}} = 0$. In order to reduce the message overload, the node waits to receive an extended table from an existing member during a fixed time T_{s}. This is the activity corresponding to the STARTUP state. If the expected reception does not happen, the node broadcasts its own *mbship_est* table and enters the NORMAL state.

In the NORMAL operation mode, each node periodically increases its heartbeat, with periodicity T_{g}, and, with probability P_{hb}, it broadcasts its local table. The probability factor P_{hb} plays the role of *fan out* parameter that decides, in a gossip algorithm for wired networks, the number of peers to be infected (only one in the original gossip–style failure detector of [97]). Such approach is not suitable for ad hoc, or in general wireless networks, since in them each communication attempt will reach only neighbours that are within communication range. Trying to reach randomly chosen peers would imply extra communication steps, and would not take advantage of the basic broadcast facility provided by wireless networks.

When another member receives the node's broadcast table, it merges the incoming information with its local table, updating entries for which higher heartbeat values have arrived, and setting T_{hb} of such entries to the current time. If the state of some node was $FAILED$ in the received table, the local entry would also be set to the failed state, and the T_{hb} value would be updated too. After processing this information, the receiver node will, with probability P_{s}, rebroadcast the updated table.

When the heartbeat of a certain node is older than a predefined time T_{f}, the node is considered to have failed, its state is changed to $FAILED$ in the local table and after a longer period T_{cleanup}, it is definitely removed from the table.

A particular case occurs when a node has been incorrectly declared as $FAILED$ by some other node. In that case it may happen that a received **Gossip** message contains an entry declaring the receiver itself as $FAILED$. Such entry is discarded when processing the incoming table, as the most up-to-date information regarding a node lies on the node itself. On the other hand, an entry declaring the local node as $CONNECTED$ but containing a *hb* value which is lower than the current value by more than a certain threshold Δ_{hb} will force the sending of a **Gossip** after processing the incoming table. This is to avoid the wrong declaration of the local node as $FAILED$ by the rest of the group due to an excessive ageing of the corresponding information. The threshold value is configurable, but in order to be useful it must satisfy $\Delta_{\mathrm{hb}} T_{\mathrm{g}} < T_{\mathrm{f}}$.

The main parameters controlling the gossip algorithm are the following.

- T_{g} is the time period between consecutive increments of the heartbeat at a given node.

- T_{f} is the maximum allowed age of the information about a particular node before setting its state to failed. The value of this parameter is chosen so that, if a member is alive, there is a high probability that a fresh heartbeat from it has arrived to any other node (from a practical point of view, $T_\mathrm{f} \geq 2T_\mathrm{g}$ is generally chosen).

- T_s is the maximal duration of the STARTUP phase, i.e. the time a foreign **Gossip** message is expected for before broadcasting the node's own singleton table and entering NORMAL operation.

- P_s is the probability that a node resends the received information (merged with its own local table) after being *infected*, i.e. after receiving a **Gossip** message.

- P_hb is the probability that a node starts spreading its local table after increasing its own heartbeat, i.e. after each T_g period.

7.4 Experimental Study

7.4.1 Observable Quantities

As already mentioned, our study focuses simultaneously on two aspects, namely the energy consumption and the quality of the membership estimation. Therefore, we need the definition of observable quantities that can be experimentally determined and yield a basis for the comparison of the various approaches.

In order to quantify the liveness of global membership information we measure the time it takes for all nodes in the system to reach a common knowledge about the connected members after a change has happened. Following previous work on characterisation of gossip algorithms in wired networks [99], this is done in two different scenarios.

- In the first one, *one joins*, the N-th node joins the group when the remaining $N-1$ are in a stable state (all of them having the same correct knowledge about present members). We measure the time elapsed from this join until the last node knows the full group composition, $T^{(\mathrm{j})}_{\mathrm{agr}}$.

- In the second scenario, *one leaves*, one node is switched off when all N nodes were in the stable state, and we measure the time it takes for the remaining members to notice the failure, $T^{(\mathrm{f})}_{\mathrm{agr}}$.

To quantify the consumption of energy, the following observable is constructed. In each execution we measure the total energy consumed by each node as a function of time. This is found to be fitted by a straight line, whose slope determines the consumed power, dE/dt, and remains approximately constant during an execution. We average this quantity over all the nodes in the network, to obtain a quantitative measure of the energetic performance of the protocols.

7.4.2 Experiments

We have performed the evaluation on two possible architectures. On one hand OLSR alone and on the other the gossip–style failure detection service described above with the support of DSR as the basic routing protocol. In the following they will be referred to as OLSR and GDSR (Gossip + DSR).

Our simulations were done using the *Network Simulator* ns2, version 2.27 [100]. The release contains a number of protocols for wireless networks, including DSR. Although the employed version of the simulator does not include an implementation of the OLSR protocol, there exist realisations developed by third parties for their use with ns2. We have used the freely downloadable one from NRL Protean Group [101]. The gossip protocol described earlier in this section was explicitly implemented to be included in the simulator.

We performed a series of experiments to measure the observables defined above, $T^{(\mathrm{j})}_{\mathrm{agr}}$, $T^{(\mathrm{f})}_{\mathrm{agr}}$ and dE/dt, as a function of the relevant protocol parameters. Each experiment consisted of a set of 20 simulations with varying network topology, whose results were later averaged.

Topologies were constructed by randomly locating the N nodes on a square grid of variable dimension, in order to test single–hop and multihop scenarios. Therefore, a grid of dimensions 200 m$\times$200 m was used for building single–hop scenarios and of size 600 m$\times$600 m for multihop scenarios. Since the range of radio signal assumed by ns2 is 250 m, in order to ensure connectivity the maximum allowed distance between a node and its nearest neighbour was 60 m for single–hop, and 225 m for multihop settings.

Each simulation was run for the *one joins* scenario to determine $T^{(\mathrm{j})}_{\mathrm{agr}}$ and dE/dt and for the *one failures* scenario to determine $T^{(\mathrm{f})}_{\mathrm{agr}}$. The experiments were repeated for different values of the total number of nodes, ranging from $N = 4$ to $N = 40$, for both proposed architectures, GDSR and OLSR. The simulation time was $\sim$ 300 s for single–hop scenarios and $\sim$ 1000 s for multihop networks, after the state of the nodes had stabilised and they shared the correct knowledge about the initial composition of the network. The parameters of ns2 determining the power consumption in reception and transmission were both fixed to 0.2 mW.

The basic factors that determine the power consumption and the liveness of global information in each case are the periodicity of information broadcasting and the lifetime of local information. The performance of both proposed approaches can only be compared when those factors are comparable.

In the case of GDSR, the relevant parameters are the gossip interval $T_{\rm g}$, the time to detect a failure, $T_{\rm f}$ and the probabilities of forwarding local or received information, $P_{\rm hb}$ and $P_{\rm s}$. In our simulations we considered $P_{\rm hb} = P_{\rm s}$, so that we had only three free parameters. In OLSR, the periodicity of information refreshing is determined by H_i and TC_i, whereas the time to detect a failure is governed by H_t and TC_t. We took $H_i = TC_i$ and $H_t = TC_t$ in our experiments, so that only two independent parameters remained. The role of H_i and TC_i in OLSR is equivalent to that of T_g/P_g in the gossip failure detection, namely the period between local broadcasts from each node. Equivalently we may say that in OLSR each node broadcasts its information $1/H_i$ times per unit time, whereas in the gossip algorithm it does so $P_{\rm g}/T_{\rm g}$ times per unit time. On the other hand, H_t and TC_t determine the expiry of information, as $T_{\rm f}$ does in the case of the gossip algorithm. This consideration allowed us to establish comparable sets of parameters for both protocols.

The value of $T_{\rm f}$ cannot be completely independent of that of $T_{\rm g}$. The probability that a given node has remained silent for a time T can be upper bounded by the probability that the basic mechanism of spontaneous gossip has not caused the sending of any **Gossip** message (i.e. ignoring the effect of foreign **Gossip** messages it receives), which can be approximately calculated as

$$P_{\rm sil} \leq \prod_{1,\dots T/T_{\rm g}} (1 - P_{\rm g}) = (1 - P_{\rm g})^{\frac{T}{T_{\rm g}}} .$$

If wrong failure detections are to be avoided, it is convenient to choose a value of $T_{\rm f}$ that ensures the probability of having a real failure is high if we have not received any message from a certain node during $T_{\rm g}$, i.e.

$$T_{\rm f} \leq T_{\rm g} \frac{\log P_{\rm sil}}{\log(1 - P_{\rm g})},$$

for a high value of $P_{\rm sil}$. If $P_{\rm g} \ll 1$ and $P_{\rm sil} \simeq 1$, the above expression can be approximated by

$$T_{\rm f} \leq \frac{T_{\rm g}}{P_{\rm g}} (1 - P_{\rm sil}) .$$

We performed a group of experiments with fixed $P_{\rm s} = 0.06$ and $T_{\rm g}$ between 1 and 5 s in GDSR, and with the corresponding values $H_i = T_{\rm g}/P_{\rm g}$ in OLSR. In the *one joins* scenario we fixed $T_{\rm f} = 2\ T_{\rm g}$ and $H_t = 6\ H_i$. In *one leaves* a more realistic $T_{\rm f} = 0.7\ T_{\rm g}/P_{\rm g}$ was chosen, as this upper–bounds the probability of detecting a fake failure due to missing gossip messages, and $H_t = 1.5\ T_{\rm g}/P_{\rm g}$ (since the implementation of OLSR requires $H_t > H_i$).

7.4.3 Results

Fig. 7.3 shows in a logarithmic scale the dependence of $T_{\mathrm{agr}}^{(\mathrm{j})}$ with T_{g} for both approaches in single and multi–hop scenarios. Notice that the value of T_{g} determines that used for H_i, the relevant OLSR parameter, through $H_i = T_{\mathrm{g}}/P_{\mathrm{s}}$. In the plots $T_{\mathrm{agr}}^{(\mathrm{j})}$ shows a linear dependence with broadcast periodicity, less pronounced in the case of GDSR, and with little sensitivity to the number of nodes in the network, N, specially in the case of OLSR. As expected, the best time to reach agreement when a new member joins is found for GDSR, with $T_{\mathrm{agr}}^{(\mathrm{j})}$ at least one order of magnitude lower than in the case of OLSR.

GDSR shows better behaviour regarding the agreement time, since its operation is optimised to react when new nodes appear. It is highly probable that a new node receives the composition of the network during its initial waiting time T_{s} and it takes only one flood to spread the complete information to the entire network. Receiving such initial message is more likely for larger N, specially in the case of a single hop. In multi–hop scenarios, higher agreement times are obtained (although still far lower than those for OLSR) as the information needs to go through a number of hops, and for constant N the probability of having a gossip within one transmission range is lower.

The time to reach agreement in the *one leaves* scenario, in fig. 7.4, shows in most cases a linear dependence with T_{g}. For GDSR it is directly governed by the time of validity of local information, given by T_{f}. It would thus be possible to tune the reaction time, at the expense of tolerating some spurious failure. In the single–hop scenario OLSR manages to reduce $T_{\mathrm{agr}}^{(\mathrm{f})}$ even under the value for GDSR. This is due to its capability of monitoring link quality, discarding links that do not show activity for long enough periods. The effect shows up when the periodicity is low enough, but it is not relevant in the multi–hop scenario.

Different results are found for energy consumption. Fig. 7.5 shows in a logarithmic scale the average consumed power with fixed $T_{\mathrm{g}} = 1$ s, as a function of the number of nodes in single and multi–hop scenarios. Contrary to what happens with liveness, the energetic performance of OLSR exceeds that of GDSR when comparable parameters are used for both protocols (starred and circle data series). The average consumption rate is lower and scales better with the number of nodes for the OLSR approach both in the case of single and multi–hop scenarios.

Summarising the results above, GDSR yields much better liveness for changes detection, whereas OLSR energy consumption is lower. Yet, the magnitude of the difference in the values obtained for $T_{\mathrm{agr}}^{(\mathrm{j})}$ is such that one may try to trade–off some liveness for energy saving. Thus, we performed a second set of experiments with GDSR, in which we limited the liveness by taking $P_s = 0.03$ while keeping the remaining parameters unchanged. The results of such tests appear also in the figures above. In fig. 7.3 the effect of lowering P_{s} in the liveness of information about joined nodes is shown (series with cross-shaped marker). We see that the result is worse than the one obtained with $P_s = 0.06$, as expected, but it is still much better than the one obtained with OLSR. In 7.5 we observe the effect of this change in energy consumption. In the single hop scenario, the OLSR approach is still more efficient, except for the lowest values of N. However, in the case of multi–hop scenario, GDSR with $P_s = 0.03$ yields a better result for all system sizes, between 4 and 40 nodes. A similar improvement is achieved in the single hop experiments by further reducing the liveness to $P_s = 0.01$, whereas the time of agreement is still better than for OLSR.

7.5 Discussion

We have simulated both considered approaches with comparable configurations. Our results show that depending on the characteristics of the system (size, single or multi–hop character), and on the requirements on the composition information (degree of liveness), either alternative may be preferable.

In particular, it is clear that GDSR may be easily tuned to provide very fast detection of joined nodes, whereas OLSR is more efficient regarding the consumption of energy. In order to get a competitive energy result with GDSR, liveness must then be relaxed. In multi–hop scenarios the balance is more easily achievable, and GDSR may provide better results in both time of agreement and consumed power. The performance of this approach could be further optimised if an on–demand approach is also adopted for the gossip service, so to compose the global estimation of the system composition only when necessary.

Furthermore, we have also run additional tests in which a constant bit rate application (i.e. one that imposes a message traffic with constant rate) is run on top of either routing service, randomly sending messages to all known members of the network at a total rate of 10 msg/s. The results indicate that the difference between OLSR and GDSR consumption reduces, but it is still possible to get better energetic results from GDSR with lower values of P_s.

It is important to notice here that our experimental results use a simple energy model (that included by the ns2 simulator) which does not reflect properly all the complexity of physical networks. Nevertheless, this preliminary study serves us to demonstrate the fact that both analysed services should not be separately studied if energy consumption and quality of service are concerned. On the contrary, the interaction of components is made evident.

Previous works studied the performance of routing protocols from the point of view of throughput or energy consumption, but to our knowledge no analysis was done of how other services could affect such results. We have explicitly shown that if DSR is chosen as a more energy–efficient routing protocol, and some estimation of the network global composition is required by user applications, the component added to provide such information may neutralise the presumed saving, and even result in a more consuming system. Thus it is not enough to study the performance of some individual component to characterise the overall performance of the system or to optimise its energy consumption. In our opinion, it is more sound to carry out a study of the performance of the set of support services that are to be deployed, in order to understand their interaction and to reach a trade–off between consumption and quality of service in a global manner.

Moreover, although our experiments have used DSR and OLSR, some of the arguments apply to reactive and proactive algorithms in a more general way. Reactive algorithms offer the advantage of not forcing a constant traffic overload, whereas proactive algorithms offer lower delays. However, when combining a routing protocol with other distributed service, we may find situations in which either approach exceeds the other's performance depending on the required QoS of the added component. If additional services are required by the applications, more components will affect the energetic performance and must be taken into account in a similar analysis.

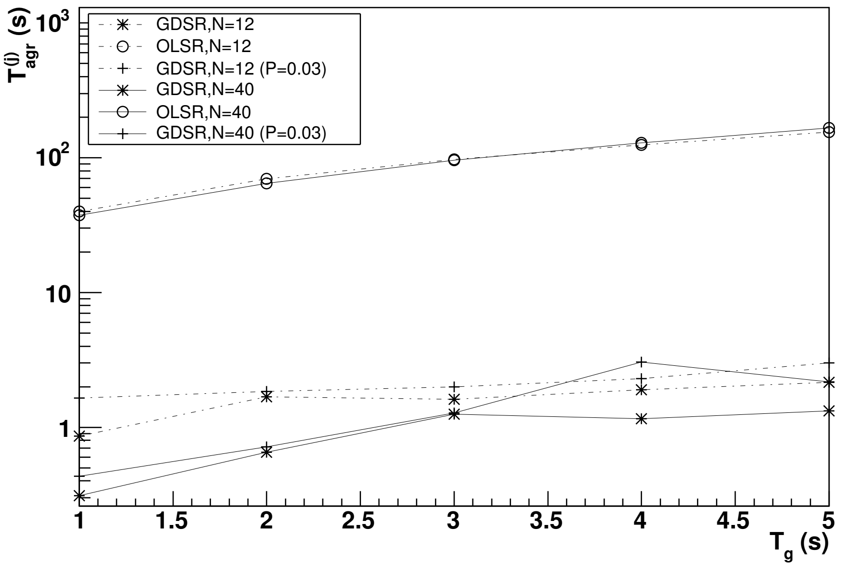
Although the energy model employed in the study is limited (we have used the one included in ns2), the results already allow some qualitative conclusions. For quantitative results, it is advisable to make use of more realistic energy models. Our analysis has been done for static scenarios since, as discussed above, these suit better our potential applications. Nevertheless, for a more complete study, it will be interesting to understand the effect of some mobility on our results.

```
algorithm gossip
type
  state_t : STARTUP, NORMAL
  entry_st_t : CONNECTED, FAILED
  mbship_entry_t : (N, st, hb, Thb)
var
  state : state_t
  mbship_est : list of mbship_entry_t
  time : local time counter
  gossip : boolean
begin
  state := STARTUP;
  time := 0;
  mbship_est := (thisNode,
    CONNECTED, 0, time);
  gossip := false;
  while true do
  case state of
    STARTUP : startup
    NORMAL : normal
  esac;
  done;
end
```

```
algorithm startup
var
  ts : timer
begin
  ts := $T_s$;
  wait for event
  case event of
  ts timeout:
    bcast Gossip(mbship_est);
    state := NORMAL;
  recv Gossip(table):
    mbship_est.merge(table);
    state := NORMAL;
  esac;
end
```

```
algorithm normal
var
  local_entry : mbship_entry_t
begin
  local_entry := mbship_est.
    getEntry(thisNode);
  if gossip then
  bcast Gossip(mbship_est);
  gossip := false;
  fi;
  wait for event
  case event of
  recv Gossip(table):
    if table contains localNode then
      tmp_entry := table.
        getEntry(thisNode);
      if local_entry.hb-tmp.hb>$\Delta_{hb}$
      then
        gossip := true;
      fi;
    fi;
    mbship_est.merge(table);
    with probability $P_s$ set
      gossip := true;
  tg timeout:
    local_entry.hb :=
      local_entry.hb+1;
    with probability $P_{hb}$ set
      gossip := true;
    for entry in mbship_est
      if entry.st=CONNECTED then
        if (time-entry.Thb)>Tfail
        then
          entry.st := FAILED;
          entry.Thb := time;
          gossip := true;
        fi;
    else
      if (time-entry.Thb)>Tcleanup
      then
        mbship_est.remove(entry);
      fi;
    fi;
    done;
  tg := $T_g$
  esac;
end
```

Figure 7.2: Basic algorithm of the gossip style membership estimation.



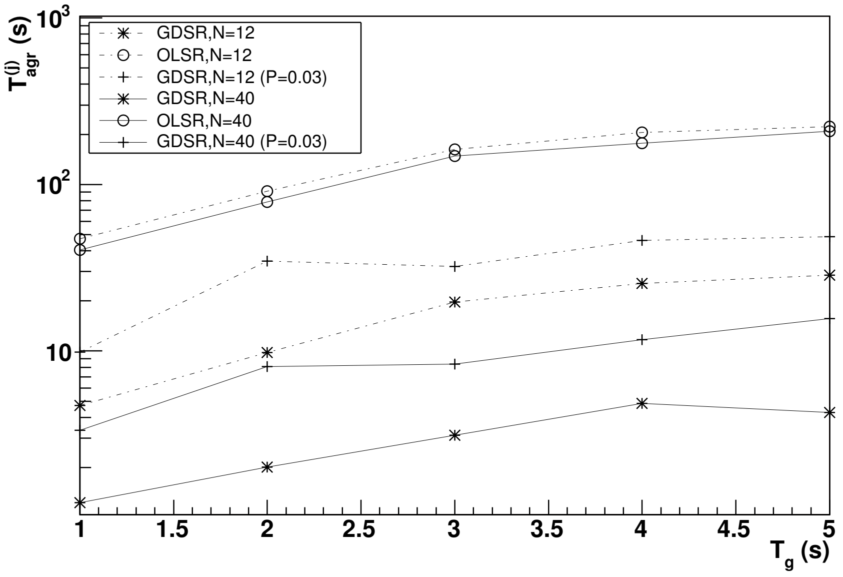
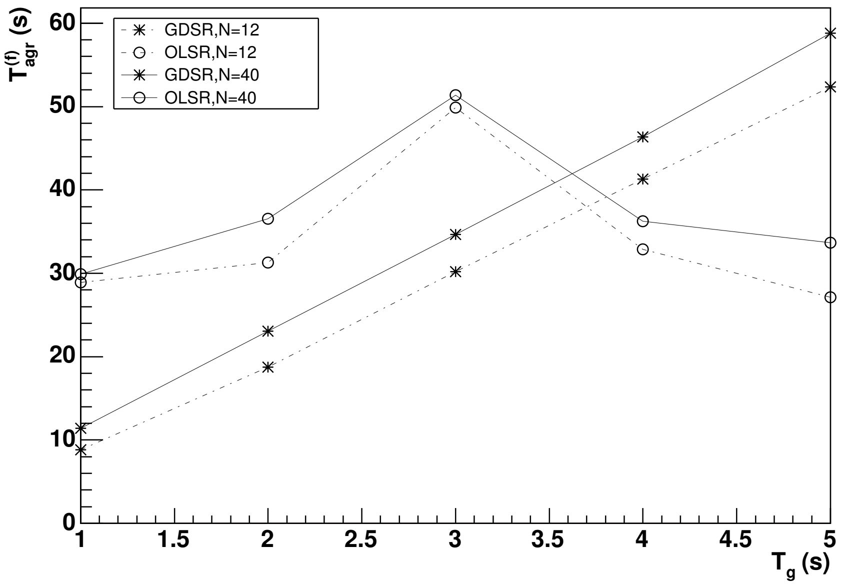


Figure 7.3: $T_{\mathrm{agr}}^{(\mathrm{j})}$ as a function of T_{g} in a single (up) and multi–hop (down) scenario.



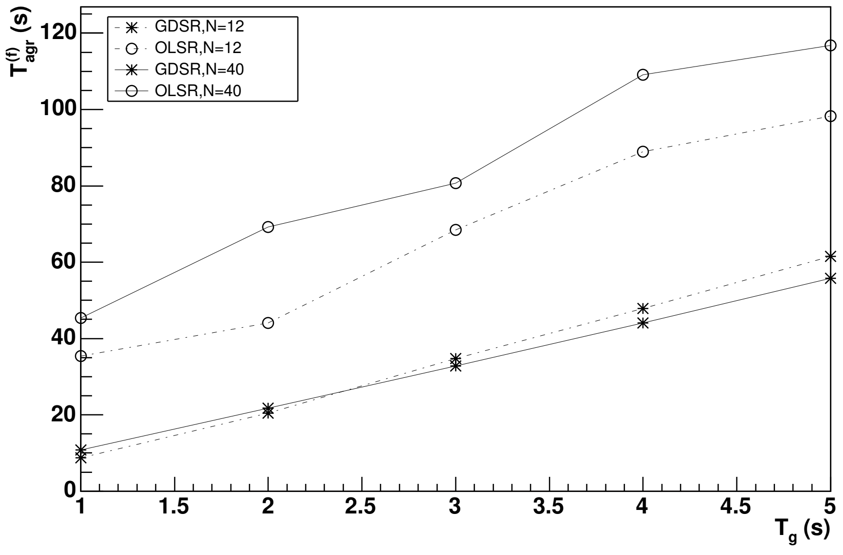
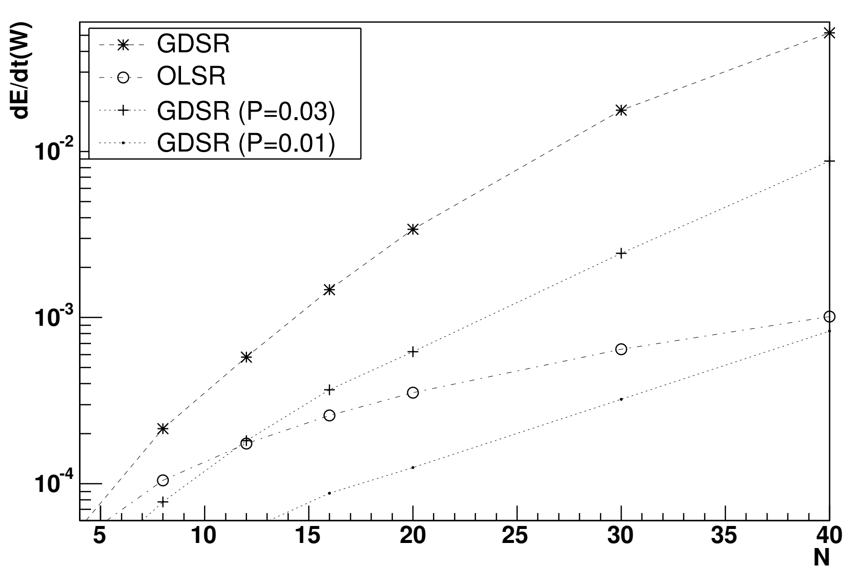


Figure 7.4: $T_{\mathrm{agr}}^{(\mathrm{f})}$ as a function of T_{g} in a single (up) and multi–hop (down) scenario.



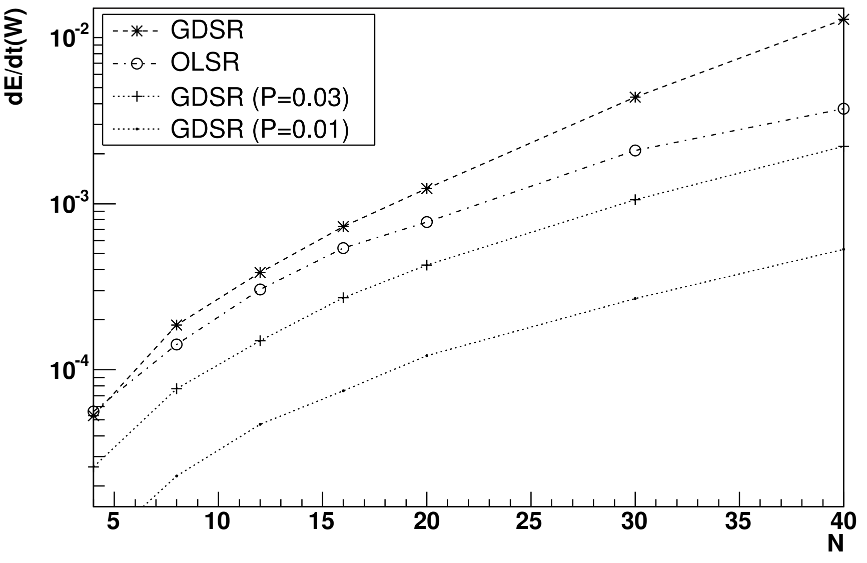


Figure 7.5: Power as a function of N in a single (up) and multi–hop (down) scenario.

Chapter 8

Conclusions

In this work we have faced the group membership problem in a broad sense. As it has been discussed, the GMP offers a fundamental support for the development of a large number of distributed services and components, hence its importance and interest. Here we have proposed a range of versions of the GMP for different distributed scenarios of great practical application nowadays. Not only specifications and usage of such services have been described, but we have also presented protocols that implement the defined services. All the proposed algorithms have been implemented, and we have executed a large number of test cases and performance measurements.

The specification of the group membership problem is not unique, but there exists a range of possible enunciates, defining variated properties to be granted by such service. Corresponding to such an assortment, there exists also a multitude of practical realisations of membership services, adapted to various environments or systems to a different degree.

The close relation between membership and group communication, due to the fundamental use of the former to support the development of the latter, have produced interdependent specifications for both services. The higher level components deployed on top of such services can base their semantics on the joint guarantees provided by the underlying services. It is the case of replication protocols of different kinds, which obtain their desired properties by making use of delivery guarantees and membership information. It is also the case of any application taking advantage of virtual synchrony or any similar semantics. However, this interrelation is not fundamental for the GMP itself. On the contrary, it can obscure its semantics to components that do not need the guarantees of group communication.

Our work is supported on two fundamental aspects.

- The theoretical cornerstone is an independent specification of the membership service, decoupled from multicast properties. Such specification is based on recent works by Chockler et al [22] and Babaoglu et al [10].

- From a practical point of view, we have opted for a modular architecture, with well–defined interfaces that regulate the interaction among different components. In particular, all the presented membership services offer a unique interface to user applications and other distributed services. This is also in agreement with recent works in the field of practical realisations of distributed services [56].

With this kind of approach, every service can be independently implemented, and exchange between different protocols is possible without the need to modify the rest of components or client applications.

Moreover, this type of architecture offers great flexibility to user applications. Depending on the requirements of a particular system, it may be decided to launch only a set of services. Even if all services are present, several applications may coexist that make use of different subsets of them.

The first part of this thesis deals with the most classical membership problem, stated for partitionable networks. We have presented the design and implementation of a new partitionable membership service for WAN–wide clusters, the HMS protocol. Despite the existence of classical membership services implementing a partitionable specification, most of them are limited in the way majority losses are handled. Some specifications do not allow the majority to completely disappear from the system, although this is a possible scenario in a real partitionable network. Even those that allow majority to be lost and recovered, leave each user application in charge of obtaining the required information regarding the last majority groups, that may be needed for the recovery tasks. Besides implementing the basic specification decoupled from group communication services (so that it can serve as basis to the development of this kind of protocols), HMS provides a new guarantee ensuring the uniformity of the majority history which as has been argued may ease the recovery phase of various user components. Although HMS is defined for monitoring the membership of a predefined set of nodes, it can be easily adapted to allow some changes to the preconfigured group. We are currently working on the definition of such an extension. This will be useful to support maintenance operations, as definitively removing a crashed node, or adding a new server to the initial group, without damaging the availability of the HMS and of services depending on it.

In the second part, more dynamical scenarios have been considered for which the classical specification of the problem is not equally suitable. We have focused in particular in the very commonly occurring environment of a large number of clients requesting services from a reduced, well–defined and interconnected group of servers. In such a system, which corresponds to common deployment scenarios for replicated services, the way clients connect to servers and the way these maintain information regarding connected clients may vary. We have specified a particular client membership service capable to maintain the most relevant information regarding connected clients to ease the development of such replicated services. Correspondingly, the client membership service also specifies the information to be delivered to clients regarding contactable servers.

We have also presented the design of the HaloMS protocol, which fulfils this specification. The protocol conforms to the modular approach of all this work, by operating on top of a basic membership service that fulfils the basic specification. In particular, we have implemented and tested it on top of the HMS protocol itself. Moreover, our implementation also conforms to the most general membership interfaces, so that information about client membership can be provided to user services with the same interface of usual view notifications. Even more, clients themselves are notified about (partial) membership of the servers group with the general interface of a membership service. Our HaloMS protocol allows client connections to be handled only by a majority server group. As a possible extension, we may consider a variation of HaloMS in which client representation is kept by minority groups after a core view change. In such case, depending on the network topology, one client might be contacted by two different partitions after core changes, and some decision should be taken by the client side of the algorithm in order to leave one of them.

The last part of this thesis approaches another type of distributed system where highly available applications may require the use of membership services. We have studied the difficulties posed by ad hoc wireless networks and come to an On–Demand specification of the membership service that suits not only this kind of networks, but also any system with a concern about bandwidth consumption. The On Demand specification of the membership service has also been realised in a definite protocol, MODUS. This provides the semantics defined by the specification based on any existing basic membership service. It has served as a basis to carry out an analysis of the On–Demand approach in terms of sent messages, i.e. communication cost of the protocol, but also for a semi–analytical analysis of the allowed message saving. The performance characterisation has been limited to the amount of sent messages, and therefore the results have to be understood as bounds to the potential saving or to the minimal membership inactivity time. In order to extract more significant measurements, the on demand approach should be specifically implemented for the scenario of interest (e.g. ad hoc networks), instead of using an off-the-shelf membership service such as HMS.

Finally, we have studied the problem of providing a membership estimation service in the context of ad hoc networks. In conjunction with a basic routing service, such a component can serve as basis for building other distributed protocols, including strictly consistent membership services or distributed consensus. We have compared two possible architectures for providing both routing and membership estimation services, from the point of view of their energy consumption and quality of the provided information on the system composition. Our simulations show that both services should be jointly analysed when optimising the consumed power and the performance, in order to reach a trade-off between QoS and energy consumption, that will always depend on the scenario and the requirements of the user applications. On top of this service, more demanding distributed services can be implemented. For instance, we can develop a specific On–Demand membership service for ad hoc networks, which improves the bounds obtained with MODUS.

The variated membership semantics presented in this thesis effectively suit the different necessities of highly available applications to be deployed on different environments. The corresponding protocols then favour and ease the development of distributed components on the various analysed scenarios. The general interfaces used for defining the interaction have shown broad enough to enclose the different semantics, while simplifying the deployment of applications that make use of the proposed protocols.

Bibliography

[1] Flaviu Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4(4):175–188, 1991.

[2] T.D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, pages 322–330, New York, USA, 1996. ACM.

[3] Aleta M. Ricciardi and Kenneth P. Birman. Using process groups to implement failure detection in asynchronous environments. In *PODC '91: Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 341–353, New York, NY, USA, 1991. ACM Press.

[4] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the formal specification of group membership services. Technical Report TR95-1534, 25, 1995.

[5] Paul D. Ezhilchelvan, Raimundo A. Macedo, and Santosh K. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *ICDCS '95: Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 296–306, Washington, DC, USA, 1995. IEEE Computer Society.

[6] Danny Dolev, Dalia Malki, and Ray Strong. A framework for partitionable membership service. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, page 343, New York, NY, USA, 1996. ACM Press.

[7] C. P. Malloth and A. Schiper. View synchronous communication in large scale distributed systems. In *Proceedings of the 2nd Open Workshop of the ESPRIT project BROADCAST (6360)*, Grenoble, France, 1995.

[8] A. Ricciardi and K. P. Birman. Process membership in asynchronous environments. Technical Report NTR 92-1328, Department of Computer Science, Cornell University, Ithaca, New York, February 1993.

[9] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, 1995.

[10] Özalp Babaoglu, Renzo Davoli, and Alberto Montresor. Group communication in partitionable systems: Specification and algorithms. *Software Engineering*, 27(4):308–336, 2001.

[11] Massimo Franceschetti and Jehoshua Bruck. A group membership algorithm with a practical specification. *IEEE Transactions on Parallel and Distributed Systems*, 12(11):1190–1200, November 2001.

[12] Bradford B. Glade, Kenneth P. Birman, Robert C.B. Cooper, and Robbert van Renesse. Light-weight process groups in the isis system. *Distributed Systems Engineering*, 1(1):29–36, 1993.

[13] Luis Rodrigues, Katherine Guo, Antonio Sargento, Robbert van Renesse, Bradford B. Glade, Paulo Veríssimo, and Kenneth P. Birman. A transparent light-weight group service. In *Symposium on Reliable Distributed Systems*, pages 130–139, 1996.

[14] Yair Amir, Claudiu Danilov, and Jonathan Robert Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, pages 327–336, Washington, DC, USA, 2000. IEEE Computer Society.

[15] Tal Anker, Gregory V. Chockler, Danny Dolev, and Idit Keidar. Scalable group membership services for novel applications. In Marios Mavronicolas, Michael Merritt, and Nir Shavit, editors, *Networks in Distributed Computing (DIMACS-97)*, DIMACS book series, pages 23–42. American Mathematical Society, 1997.

[16] Idit Keidar, Jeremy Sussman, Keith Marzullo, and Danny Dolev. Moshe: A group membership service for wans. *ACM Trans. Comput. Syst.*, 20(3):191–238, 2002.

[17] K. Berket, D. Agarwal, and O. Chevassut. A practical approach to the intergroup protocols. *Future Generation Computer Systems*, 18(5):709–719, 2002.

[18] Luis Irún-Briz, Hendrik Decker, Rubén de Juan-Marín, Francisco Castro-Company, Jose E. Armendáriz-I nigo, and Francesc D. Mu noz Escoí. Madis: A slim middleware for database replication. In *Euro-Par 2005 Parallel Processing: 11th International Euro-Par Conference*, volume 3648 of *Lecture Notes in Computer Science*, pages 349–359. Springer-Verlag, 2005.

[19] M. Jandl, A. Szep, R. Smeikal, and K.M. Goeschka. Dedisys: A european union aided research project. In *Proc. Companion 5th ACM/IFIP/USENIX Int'l Conf. on Middleware*, page 330, 2004. See also the project webpage, http:/www.dedisys.org.

[20] http://www.iti.es/groups/sifi.

[21] Ozalp Babaoglu and André Schiper. On group communication in large-scale distributed systems. In *ACM SIGOPS European Workshop*, pages 17–22, 1994.

[22] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[23] D. Dolev, D. Malki, and R. Strong. An asynchronous membership protocol that tolerates partitions. Technical report, The Hebrew University of Jerusalem, 1993.

[24] Rachid Guerraoui and André Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41, 2001.

[25] Rachid Guerraoui, Michel Hurfin, Achour Mostefaoui, Riucarlos Oliveira, Michel Raynal, and André Schiper. Consensus in asynchronous distributed systems: A concise guided tour. In *Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems*, pages 33–47, London, UK, 1999. Springer-Verlag.

[26] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[27] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[28] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[29] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[30] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. Technical Report TR85-694, Department of Computer Science, Cornell University, 1985.

[31] Roy Friedman and Robert van Renesse. Strong and weak virtual synchrony in horus. In *SRDS '96: Proceedings of the 15th Symposium on Reliable Distributed Systems (SRDS '96)*, page 140, Washington, DC, USA, 1996. IEEE Computer Society.

[32] L.E. Moser, Y. Amir, P.M. Melliar-Smith, and D.A. Agarwal. Extended virtual synchrony. In *The 14th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, 1994.

[33] Jeremy Sussman, Idit Keidar, and Keith Marzullo. Optimistic virtual synchrony. In *SRDS '00: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, page 42, Washington, DC, USA, 2000. IEEE Computer Society.

[34] R. Van Renesse et al. Horus: A flexible group communications system. Technical Report TR95-1500, 23, 1995.

[35] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, 1996.

[36] D.A. Agarwal et al. The Totem multiple-ring ordering and topology maintenance protocol. *ACM Transactions on Computer Systems*, 16(2):93–132, 1998.

[37] Özalp Babaoglu, Renzo Davoli, Luigi-Alberto Giachini, and Mary Gray Baker. Relacs: A communications infrastructure for constructing reliable applications in large-scale distributed systems. In *Proc. of the 28th Hawaii Int. Conf. on System Sciences, HICSS (2)*, pages 612–621. IEEE Computer Society, 1995.

[38] Ryan Caudy. Scalable process group membership for the spread toolkit, 2004.

[39] Tal Anker, Gregory V. Chockler, I. Shnaiderman, and Danny Dolev. The design of xpand: A group communication system for wide area networks. Technical Report 2000-31, Institute of Computer Science, Hebrew University, Jerusalem, Israel, 2000.

[40] Richard A. Golding and Kim Taylor. Group membership in the epidemic style. Technical Report UCSC-CRL-92-13, University of California at Santa Cruz, 1992.

[41] Richard A. Golding. A weak-Consistency Architecture For Distributed Information Services. *Computing Systems*, 5(4):379–405, 1992. Technical Report UCSC-CRL-92-31.

[42] Roy Friedman. Fuzzy group membership. In *Future Directions in Distributed Computing*, pages 114–118, 2002.

[43] James S. Pascoe, Roger J. Loader, and Vaidy S. Sunderam. Collaborative group membership. *J. Supercomput.*, 22(1):55–68, 2002.

[44] Tal Anker, Gregory Chockler, Danny Dolev, and Idit Keidar. The caelum toolkit for CSCW: The sky is the limit. In Abraham Silberschatz and Peretz Shoval, editors, *Next Generation Information Technologies and Systems (NGITS '97), Third International Workshop, Neve Ilan, Israel, 1997*, 1997.

[45] Jianjun Zhang, Ling Liu, Calton Pu, and Mostafa Ammar. Reliable peer-to-peer end system multicasting through replication. In *P2P'04: Proceedings of the Fourth International Conference on Peer-to-Peer Computing (P2P'04)*, pages 235–242, Washington, DC, USA, 2004. IEEE Computer Society.

[46] Mark Jelasity, Wojtek Kowalczyk, and Maarten van Steen. Newscast computing. Technical Report IR-CS-006, Vrije Universiteit Amsterdam, Department of Computer Science, November 2003.

[47] Spyros Voulgaris, Márk Jelasity, and Marteen van Steen. A robust and scalable peer-to-peer gossiping protocol. In *Agents and Peer-to-Peer Computing: Second International Workshop*, volume 2872 of *Lecture Notes in Computer Science*, pages 47–58, 2004.

[48] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.

[49] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In *Third International Workshop on Networked Group Communications (NGC 2001), London, UK*, 2001.

[50] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2), February 2003.

[51] Linda Briesemeister and Gnter Hommel. Localized group membership service for ad hoc networks. In *International Workshop on Ad Hoc Networking (IWAHN)*, pages 94–100, August 2002.

[52] Qingfeng Huang, Christine Julien, and Gruia-Catalin Roman. Relying on safe distance to achieve strong partitionable group membership in ad hoc networks. *IEEE Transactions on Mobile Computing*, 3(2):192–205, 2004.

[53] Specifying and using a partitionable group communication service. *ACM Trans. Comput. Syst.*, 19(2):171–216, 2001.

[54] Özalp Babaoglu, Renzo Davoli, and Alberto Montresor. Group membership and view synchrony in partitionable asynchronous distributed systems: Specifications. *Operating Systems Review*, 31(2):11–22, 1997.

[55] André Schiper and Sam Toueg. From set membership to group membership: A separation of concerns. Technical Report IC-2003/56, 2003.

[56] Matthias Wiesmann, Xavier Défago, and André Schiper. Group communication based on standard interfaces. In *2nd IEEE Intl. Symp. on Network Computing and Applications (NCA-03)*, 2003.

[57] B. Ban. Design and implementation of a reliable group communication toolkit for java, September 1998.

[58] H. Miranda, A. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *Proc. 21st International conference on Distributed Computing Systems (ICDCS-21)*, pages 707–710, 2001.

[59] Matti A. Hiltunen and Richard D. Schlichting. The Cactus approach to building configurable middleware services. In *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, 2000.

[60] André Schiper. Failure detection vs group membership in fault-tolerant distributed systems: Hidden trade-offs. In *PAPM-PROBMIV '02: Proceedings of the Second Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification*, pages 1–15, London, UK, 2002. Springer-Verlag.

[61] Yolanda Tomás, Raúl Salinas, and Pablo Galdámez. Tigerweb: Una herramienta para el análisis de la seguridad perimetral en redes ip. In *Actas del I Congreso Espaol de Informtica*, 2005.

[62] M.C. Bañuls and P. Galdámez. Extended membership problem for open groups: Specification and solution. In M. Daydé et al., editor, *VECPAR 2004: High Performance Computing for Computational Science*, number 3402 in LNCS, pages 288–301. Springer–Verlag, 2005.

[63] Mari Carmen Bañuls and Pablo Galdámez. Membership service for open clusters. In *Concurrencia y Sistemas Distribuidos*, page 39. Publicacions de la Universitat Jaume I, 2003. Actas de las XI Jornadas de Concurrencia.

[64] Nancy A. Lynch and Mark Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989. Also, Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology.

[65] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987.

[66] M.C. Bañuls and P. Galdámez. Technical Report ITI-ITE-03/01, Instituto Tecnológico de Informática, Univ. Politécnica Valencia, 2003. http://www.iti.upv.es/sifi/en/publications/2003/index.html.

[67] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Database replication techniques: a three parameter classification. In *Proceedings of 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*, Nürenberg, Germany, 2000. IEEE Computer Society.

[68] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

[69] Mari Carmen Bañuls and Pablo Galdámez. Halo membership service: A specific membership service for large dynamic client groups. In *Actas de las XII Jornadas de Concurrencia y Sistemas Distribuidos*, pages 125–134. Universidad Rey Juan Carlos, 2004.

[70] Mari Carmen Bañuls and Pablo Galdámez. Client group membership as an architectural approach for dependability in large scale systems. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05)*, page 291b. IEEE Computer Society, Los Alamitos, CA, USA, 2005.

[71] C.T. Karamanolis and J.N. Magee. Configurable highly available distributed services. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems*, pages 118–127.

[72] C.T. Karamanolis and J.N. Magee. Client access protocols for replicated services. *Software Engineering*, 25(1):3–21, 1999.

[73] Ozalp Babaoğlu and André Schiper. On group communication in large-scale distributed systems. *SIGOPS Operating Systems Review*, 29(1):62–67, 1995.

[74] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[75] Mari Carmen Bañuls and Pablo Galdámez. On demand membership service for energy–aware networks. In *Proceedings of the 16th International Conference on Database and Expert Systems and Applications, DEXA 2005, Copenhagen (Denmark)*, pages 315–319. IEEE Computer Society, 2005.

[76] Mari Carmen Bañuls and Pablo Galdámez. On-demand membership service for energy-aware networks. In *Actas del I Congreso Español de Informtica (XIII Jornadas de Concurrencia y Sistemas Distribuidos)*, pages 227–237. Thomson Paraninfo, 2005.

[77] S. Corson and J. Macker. Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations, 1999.

[78] Internet engineering task force manet working group charter.

[79] E. Miedes, M. C. Bañuls, and P. Galdámez. Comparando protocolos mediante javagroups. Technical report, Instituto Tecnologico de Informatica, 2003. Available from http://www.iti.upv.es/groups/sifi/papers/2003.

[80] I. Keidar. Challenges in evaluating distributed algorithms. In André Schiper, A.A. Shvartsman, H. Weatherspoon, and B.Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *LNCS*, page 40. Springer-Verlag, 2003.

[81] J. C. García, M.C. Bañuls, and P. Galdámez. Trading off consumption of routing and precision of membership. In *Proceedings of the 3rd Intn't Conference Communications and Computer Networks, October 24–26, Marina del Rey, CA (USA)*, pages 108–113. ACTA Press, 2005.

[82] David B. Johnson, David A. Maltz, and Josh Broch. *DSR: The Dynamic Source Routing Protocol for Multihop Wireless Ad Hoc Networks*, chapter 5, pages 139–172. Addison-Wesley, 2001.

[83] Charles E. Perkins and Elizabeth M. Royer. Ad-hoc on-demand distance vector routing. In *WMCSA '99: Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, page 90, Washington, DC, USA, 1999. IEEE Computer Society.

[84] Vincent D. Park and M. Scott Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *INFOCOM '97: Proceedings of the INFOCOM '97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution*, page 1405, Washington, DC, USA, 1997. IEEE Computer Society.

[85] P. Jacquet, P. Mühlethaler, T. Clausen, A. Laouiti, A. Qayyum, and L. Viennot. Optimized link state routing protocol for ad hoc networks. In *Proceedings of the 5th IEEE Multi Topic Conference (INMIC 2001)*, 2001.

[86] Charles E. Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. In *SIGCOMM '94: Proceedings of the conference on Communications architectures, protocols and applications*, pages 234–244, New York, NY, USA, 1994. ACM Press.

[87] Juan-Carlos Cano and Pietro Manzoni. A performance comparison of energy consumption for mobile ad hoc network routing protocols. In *MASCOTS'00: Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 57–64, Washington, DC, USA, 2000. IEEE Computer Society.

[88] Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu, and Jorjeta Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Mobile Computing and Networking*, pages 85–97, 1998.

[89] Thomas Heide Clausen, Gitte Hansen, Lars Christensen, and Gehrd Behrmann. The optimized link state routing protocol, evaluation through experiments and simulation. In *The 4th International Symposium on Wireless Personal Multimedia Communications*, September 2001.

[90] Optimized link state routing protocol (olsr), 2003.

[91] B. Baker and R. Shostak. Gossips and telephones. *Discrete Mathematics*, 2:191–193, 1972.

[92] Patrick T. Eugster, Rachid Guerraoui, Ane-Marie Kermarrec, and Laurent Massoulié. Epidemic information dissemination in distributed systems. *IEEE Computer*, pages 60–67, May 2004.

[93] P. Th. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A. M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, 2003.

[94] S. Voulgaris, M. Jelasity, and M. van Steen. A robust and scalable peer-to-peer gossiping protocol. In *Proc. 2nd Int'l Workshop on Agents and Peer-to-Peer Computing (AP2PC 2003), Melbourne, Australia*, 2003.

[95] M. Jelasity and O. Babaoglu. T-MAN: Fast gossip-based construction of large-scale overlay topologies. Technical Report UBLCS 2004-7, Univ. Bologna, 2004.

[96] I. Gupta, R. van Renesse, and K. Birman. Scalable fault-tolerant aggregation in large process groups. In *Proc. Conf. on Dependable Systems and Networks*, 2001.

[97] R. Van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proceedings of Middleware'98*, pages 55–70. IFIP, The Lake District, UK, 1998.

[98] Zygmunt J. Haas, Joseph Y. Halpern, and Erran L. Li. Gossip-based ad hoc routing. In *INFOCOM*, 2002.

[99] Mark W. Burns, Alan D. George, and Bradley A. Wallace. Simulative performance analysis of gossip failure detection of scalable distributed systems. *Cluster Computing*, (2):201–217, 1999.

[100] The VINT project. The NS2 manual. Technical report, ISI, 2004. http://www.isi.edu/nsnam/ns/ns-documentation.html.

[101] NRL Protean Group. http://pf.itd.nrl.navy.mil.

[102] Nancy Lynch and Frits Vaandrager. Forward and backward simulations for timing-based systems. In *Proceedings of Real-Time: Theory in Practice (REX Workshop, Mook, The Netherlands, June 1991)*, volume 600 of *Lecture Notes in Computer Science*, pages 397–446. Springer-Verlag, 1992.

[103] Nancy Lynch and Frits Vaandrager. Forward and backward simulations, part ii: Timing-based systems. *Information and Computation*, 128(1):1–25, 1996.

Appendix A

Proof of Correctness of the Presented Protocols

Each one of the three main protocols presented in this work is designed to fulfil a particular formal specification expressed as a list of properties. In this appendix we sketch, for each of the three main algorithms presented, the proof of some of those properties, and give a pseudo-formal proof for the non-trivial ones.

A.1 HMS Protocol

The formal specification the HMS protocol satisfies was presented in sect. 3.1.2 as the basic GMP.

A.1.1 Notation

The system model we consider is formed by a set of processes with static identifiers that communicate by exchanging messages. The communication network implements `send` and `receive` primitives, and the system is asynchronous, since no bound exists on network delays or processors relative speed. Each process can be modelled as a finite state automaton.

We use the notation $a \prec_p b$ to indicate that *event a causally precedes event b in processor p.* The precedence shown by $\prec$ is immediate, while closure of this relation will be denoted by $\overset{*}{\prec}$. The $\prec$ symbol with no subindex will be used to denote causal precedence between events that occur in different nodes.

The relevant events for the correctness proof are those related to the different stages of view confirmation. Some of them correspond also to the delivery of information, in the form of membership events, to external observers, i.e. higher level applications or components that act as membership listeners. The following table describes them and summarises the used notation. For each entry, the subindex indicates the process where the event takes place.

event	*description*
$send_p(msg)$	Sending of message msg by processor p.
$receive_p(msg)$	Reception of message msg at processor p.
$\texttt{prepare}_p(V_\mathrm{i})$	View V_i is added to the view list with flag `prepared`. When the service is implemented according to the generic interfaces, a MBSHIP_CHANGE event is produced.
$\texttt{discard}_p(V_\mathrm{i})$	View V_i is discarded and removed from the list.
$\texttt{commit}_p(V_\mathrm{i})$	View V_i is flagged `committed`. A MBSHIP_VIEW event is produced and notified if the corresponding listeners exist.
$\texttt{release}_p(V_\mathrm{i})$	View V_i is flagged `released`. A MBSHIP_RUNNING event is produced for membership listeners.
$\texttt{upcommit}_p(V_\mathrm{i})$	Majority view V_i is saved to the majority history. A MBSHIP_UPVIEW event is produced and notified if the corresponding listeners exist.

Besides the precedence relation for events, we also introduce a higher level precedence, that of views. Thus we will say that view V_j is the immediate successor of view V_i in processor N_p, and write $V_\mathrm{i} \overset{(1)}{<}_p V_\mathrm{j}$, if

$$\texttt{commit}_p(V_\mathrm{i}) \overset{*}{\prec} \texttt{commit}_p(V_\mathrm{j})$$

and

$$\nexists\ V_\mathrm{k} \quad \text{s.t.} \quad \texttt{commit}_p(V_\mathrm{i}) \overset{*}{\prec} \texttt{commit}_p(V_\mathrm{k}) \overset{*}{\prec} \texttt{commit}_p(V_\mathrm{j}).$$

If the precedence is not immediate, we will write $V_\mathrm{i} \overset{*}{<}_p V_\mathrm{j}$.

For majority views, a total order relation is defined. Therefore we will write $V_\mathrm{i} \overset{(1)}{<} V_\mathrm{j}$ when V_j is the view that immediately follows V_i in the (global) majority history.

A.1.2 Safety Properties

Self Inclusion (Property GM.1)

This property requires a process to be member of each view it installs and is trivially satisfied by our algorithm. The node is present in the initial *zero* view installed at startup (see sect. 4.3.2). When the node is alive, it can only receive and process proposals which contain it. Notice that **Setmem** and **Step** messages are only addressed to members of the corresponding view, and the master which issues such messages is always contained in its own proposal. Therefore the node must be present in any view it knows about, capable to be installed.

Initial View Event (Property GM.2)

The requirement that any node starts in a stable *zero* configuration with itself as the only member also ensures than any later event such as message reception or sending will take place within a well-defined view. This obviously applies to view oriented systems only, where the membership service is necessarily launched at startup. Fos such systems, the *zero* configuration of HMS would play the role of the required initial view.

Local Monotonicity (Property GM.3)

The Local Monotonicity property requires that the order of installation of common views is respected among nodes that share them. Since our protocol is not implemented on top of any group communication service, it takes advantage of no guarantee on the order of delivery of messages. Therefore this property must be enforced by the algorithm itself.

Proposition 1. (Previous result) *Given a node N_p and two views V_{i} and V_{j} committed in this order by N_p, then the `commit` action for the first one precedes the `prepare` action for the second.*

$$\texttt{commit}_p(V_{\mathrm{i}}) \overset{*}{\prec}_p \texttt{commit}_p(V_{\mathrm{j}}) \Rightarrow \texttt{commit}_p(V_{\mathrm{i}}) \overset{*}{\prec}_p \texttt{prepare}_p(V_{\mathrm{j}}). \tag{A.1}$$

Proof. Since in a given node, N_{p}, local `prepare` action always precedes local `commit` of the corresponding view, we may write

$$\texttt{prepare}_p(V_{\mathrm{i}}) \overset{*}{\prec}_p \texttt{commit}_p(V_{\mathrm{j}}).$$

We will now show that

$$\texttt{prepare}_p(V_{\mathrm{i}}) \overset{*}{\prec}_p \texttt{prepare}_p(V_{\mathrm{j}}). \tag{A.2}$$

To see this, let us assume that the opposite is true, i.e. that

$$\texttt{prepare}_p(V_{\mathrm{j}}) \overset{*}{\prec}_p \texttt{prepare}_p(V_{\mathrm{i}}).$$

In that case, either

$$\texttt{discard}_p(V_{\mathrm{j}}) \overset{*}{\prec}_p \texttt{prepare}_p(V_{\mathrm{i}})$$

or

$$\texttt{commit}_p(V_\mathrm{j}) \overset{*}{\prec}_p \texttt{prepare}_p(V_\mathrm{i})$$

hold, since V_j view would enter the view list before V_i and would then be processed according to the algorithm in sect. 4.3.2. Notice that the list cannot hold two prepared views, as the algorithm ensures that when adding a `prepared` view, any older `prepared` one is discarded, unless the addition is preceded by the `commit` of the previously `prepared` one (e.g. if the **Setmem** message contained the confirmation). Both statements contradict our hypothesis, the first one because it implies no $\texttt{commit}_p(V_\mathrm{j})$ event will occur in N_p, the second because directly violates the hypothesis in (A.1). Then (A.2) must hold.

By a similar argument, if $\texttt{prepare}_p(V_\mathrm{j}) \overset{*}{\prec}_p \texttt{commit}_p(V_\mathrm{i})$, and taking into account (A.2), V_i would be discarded before executing $\texttt{prepare}_p(V_\mathrm{j})$. So

$$\texttt{commit}_p(V_\mathrm{i}) \overset{*}{\prec}_p \texttt{prepare}_p(V_\mathrm{j}).$$

□

Proposition 2. (Property GM.3) *Given two views V_i, V_j that contain a common node, N_p, and satisfy*

$$\texttt{commit}_p(V_\mathrm{i}) \overset{*}{\prec}_p \texttt{commit}_p(V_\mathrm{j}), \tag{A.3}$$

then

$$\texttt{commit}_q(V_\mathrm{i}) \overset{*}{\prec}_q \texttt{commit}_q(V_\mathrm{j}) \tag{A.4}$$

holds for any other node N_q, contained in V_i, V_j that installs both views, i.e. that executes both `commit` *events.*

Proof. (Property GM.3)

Let us consider the case when neither N_p nor N_q does leave the group between V_i and V_j, i.e. no *Partition Merging* procedure is run in between (no **Joined** or **Update** messages are received by N_q). In such case, the following arguments apply both to majority and minority views. Let N_i and N_j be the masters of V_i and V_j respectively. Then, the following relations hold $\forall N_s \in V_\mathrm{i}$ such that N_s remains alive in V_j:

$$\texttt{prepare}_i(V_\mathrm{i}) \overset{*}{\prec} \texttt{prepare}_s(V_\mathrm{i}); \tag{A.5}$$

$$\texttt{prepare}_s(V_\mathrm{i}) \overset{*}{\prec} \texttt{commit}_i(V_\mathrm{i}); \tag{A.6}$$

$$\texttt{commit}_i(V_\mathrm{i}) \overset{*}{\prec} \texttt{commit}_s(V_\mathrm{i}). \tag{A.7}$$

In an analogous way, $\forall N_s \in V_\mathrm{j}$ such that N_s remains alive in V_j:

$$\texttt{prepare}_j(V_\mathrm{j}) \overset{*}{\prec} \texttt{prepare}_s(V_\mathrm{j}); \tag{A.8}$$

$$\texttt{prepare}_s(V_\mathrm{j}) \overset{*}{\prec} \texttt{commit}_j(V_\mathrm{j}); \tag{A.9}$$

$$\texttt{commit}_j(V_\mathrm{j}) \overset{*}{\prec} \texttt{commit}_s(V_\mathrm{j}). \tag{A.10}$$

On the other hand, both N_p and N_q remain correct (we are assuming they stay in the group), and by proposition **1** we have

$$\texttt{commit}_p(V_\mathrm{i}) \overset{*}{\prec}_p \texttt{prepare}_p(V_\mathrm{j}).$$

Therefore, by combining (A.5), (A.6) and (A.10) applied to N_q and (A.7), (A.1) and (A.9) applied to N_p we get

$$\texttt{prepare}_q(V_\mathrm{i}) \overset{*}{\prec}_q \texttt{commit}_q(V_\mathrm{j}). \tag{A.11}$$

We will see now that the thesis

$$\texttt{commit}_q(V_\mathrm{i}) \overset{*}{\prec}_q \texttt{commit}_q(V_\mathrm{j})$$

must hold.

Let us assume that the opposite is true, i.e.

$$\texttt{commit}_q(V_\mathrm{j}) \overset{*}{\prec}_q \texttt{commit}_q(V_\mathrm{i}).$$

Then, by the same reasoning we would conclude that

$$\texttt{prepare}_p(V_\mathrm{j}) \overset{*}{\prec}_p \texttt{commit}_p(V_\mathrm{i}),$$

which contradicts (A.1). Therefore, the thesis (A.4) must be true.

Let us now consider the case, contrary to our initial assumption, when the *Partition Merging* protocol is executed between both views. Since this protocol does not spread information on minority views, the only way for both N_p and N_q to commit a given minority view is that they take part of the same group simultaneously. In that case the order of the `commit` events will be necessarily the same in both nodes.

Therefore we are only left to consider here the case in which both views are in majority and a partition and merging happens in between. Let us assume then that N_q fails or abandons the group between V_i and V_j. Since by hypothesis N_q executes the $\texttt{commit}(V_\mathrm{i})$ event, and that is only possible as a response to the master's **Step** message (or to an **Ends** message if N_q itself was the master), then the `commit` event in N_q clearly precedes any other `commit` occurred while in the minority, and thus also precedes the $\texttt{commit}_q(V_\mathrm{j})$ event. □

View Agreement (Property GM.4)

Proposition 3. (Property GM.4 (i)) *Let N_p be a node that installs a view V_j as the immediate successor of V_i, i.e. $V_\mathrm{i} \overset{(1)}{<}_p V_\mathrm{j}$. Then*

$$\forall N_q \in V_\mathrm{i} \cap V_\mathrm{j}, \quad \texttt{commit}_q(V_\mathrm{i}) \overset{*}{\prec} \texttt{commit}_p(V_\mathrm{j}).$$

Proof. Let us consider N_p such that $V_{\mathrm{i}} \overset{(1)}{<_p} V_{\mathrm{j}}$, and let N_i and N_j be the masters of V_{i} and V_{j}, respectively.

1. If $N_i \equiv N_j$, i.e. no master change has taken place, then

$$\mathtt{commit}_i(V_{\mathrm{i}}) \overset{*}{\prec_i} \mathtt{prepare}_i(V_{\mathrm{j}}) \overset{*}{\prec_i} \mathtt{commit}_i(V_{\mathrm{j}}) \overset{*}{\prec} \mathtt{commit}_p(V_{\mathrm{j}}),$$

   and the proposal of V_{j} by N_j contains also the confirmation of V_{i}. Then

$$\mathtt{commit}_q(V_{\mathrm{i}}) \overset{*}{\prec_q} \mathtt{prepare}_q(V_{\mathrm{j}}) \overset{*}{\prec} \mathtt{commit}_i(V_{\mathrm{j}}), \quad \forall N_q \in V_{\mathrm{i}} \cap V_{\mathrm{j}}.$$

   The **Setmem** message does not contain the confirmation of V_{i} only if it was already released,

$$\mathtt{release}_i(V_{\mathrm{i}}) \overset{*}{\prec_i} \mathtt{prepare}_i(V_{\mathrm{j}}),$$

   in which case $\mathtt{commit}_q(V_{\mathrm{i}}) \overset{*}{\prec} \mathtt{commit}_p(V_{\mathrm{j}})$ is trivially deduced, since `commit` in members precedes `release` in the master.

2. If masters differ then there are two possibilities.

   - There has been a partition merging between both views, such that V_{i} was a minority and now N_p has been accepted by a larger group, so that the next view, V_{j} is a different one. Since N_i has executed the installation of V_{j} in V_{i} (there has been no committed view in between),

$$\mathit{receive}_p(\mathbf{Joined}(V_{\mathrm{M}}, V_{\mathrm{i}})) \overset{*}{\prec_p} \mathit{send}_p(\mathbf{Ready}(V_{\mathrm{M}})) \overset{*}{\prec_p} \mathtt{commit}_p(V_{\mathrm{j}}).$$

     Since $N_q \in V_{\mathrm{i}} \cap V_{\mathrm{j}}$,

$$\mathit{receive}_q(\mathbf{Joined}(V_{\mathrm{M}}, V_{\mathrm{i}})) \overset{*}{\prec_q} \mathit{send}_q(\mathbf{Ready}(V_{\mathrm{M}})) \overset{*}{\prec_q} \mathtt{prepare}_q(V_{\mathrm{j}}) \overset{*}{\prec} \mathtt{commit}_p(V_{\mathrm{j}}).$$

     But the **Joined** message carries the reference of the minority view it pretends to accept, V_{i}, and N_q will not process it unless it agrees with its current view, then

$$\mathtt{commit}_q(V_{\mathrm{i}}) \overset{*}{\prec_q} \mathit{receive}_q(\mathbf{Joined}(V_{\mathrm{M}}, V_{\mathrm{i}})).$$

     If N_i had been in the (relatively) majority partition, a master changing procedure must necessarily be run for a different master to lead the next view. Since the new master is including merged nodes, this means that the failed one (the master of the majority group) had already sent its extended proposal, and the same reasoning applies before the failure.

   - There has been no merging between the proposals of V_{i} and V_{j}. Then a master change must have taken place. If N_p has executed the $\mathtt{commit}(V_{\mathrm{i}})$ event before the change, N_j will have collected that information during the master change subprotocol, from the **View** message. Otherwise, the confirmation order has been sent by N_j itself, after collecting all the answers to **Change** and determining that V_i was committed already by some of the survivors. In any case, the **Setmem** message containing the confirmation of V_i and the proposal of V_j will be addressed, in particular to all $N_q \in V_\mathrm{i} \cap V_\mathrm{j}$, and

$$\mathtt{commit}_q(V_\mathrm{i}) \overset{*}{\prec}_q \mathtt{prepare}_q(V_\mathrm{j}) \overset{*}{\prec} \mathtt{commit}_j(V_\mathrm{j}) \overset{*}{\prec} \mathtt{commit}_p(V_\mathrm{j}).$$

It might also happen that another master change takes place before N_p commits V_j. In such case it is also easy to argue that later confirmation of V_j is only possible if all its (surviving) members had already executed the `prepare` action, and thus committed V_i.

$\square$

Proposition 4. (Previous result) Let N_p execute a prepare action for some view, $\mathtt{prepare}_p(V_1)$. Then it eventually performs a commit action, i.e.

$$\exists V_k \quad s.t. \quad \mathtt{prepare}_p(V_1) \overset{*}{\prec}_p \mathtt{commit}_p(V_k).$$

Proof. Let N_1 be the master of view V_1. Then, according to A.5 it has executed its own `prepare` action before N_p's. The basic algorithm of view proposal and confirmation, together with reliable multisending and the failure detector ensure that either N_1 finishes the confirmation of V_1 until the `release` action or a failure is detected and a new view, V_2, is proposed by either N_1 or by a new master. In the first case, $\mathtt{commit}_p(V_1)$ occurs. In the second, we may have two possibilities.

1. $N_p \in V_2$, in which case the same reasoning applies again;
2. $N_p \notin V_2$, and eventually N_p will confirm that view (or a subsequent one). The exclusion mechanism (see sect. 4.2.5) ensures that N_p will receive a notification allowing it to exclude N_1 and all other nodes excluding itself. If N_p is not the only excluded node, the corresponding substitute master for the excluded group will be in charge of reacting to such notification, propose a new view and act as the master thereon, thus granting the eventual confirmation of a view with or without N_p. If N_p is excluded by every possible substitute master, then it is left on itself and will react to the failure notifications of the exclusion mechanism by installing a singleton view, thus executing $\mathtt{commit}_p$.

$\square$

Proposition 5. (Property GM.4 (ii)) Let V_i be a committed view, and let $N_q \in V_\mathrm{i}$ such that q never installs V_i or q installs V_k as an immediate successor to V_i, $V_\mathrm{i} \overset{(1)}{<}_q V_\mathrm{k}$. Then p installs a different view V_j immediately succeeding V_i, i.e. $V_\mathrm{i} \overset{(1)}{<}_p V_\mathrm{j}$.

Proof. If N_i is the master of $V_{\rm i}$

$$\texttt{commit}_i(V_{\rm i}) \overset{*}{\prec} \mathit{send}_i(\mathbf{Step(1)}) \overset{*}{\prec} \texttt{commit}_p(V_{\rm i}),$$

then the master has sent the confirmation to all members, in particular to N_q. Since the sending is retried by the mechanism that makes point–to–point messages reliable, if N_q does not commit the view, one of the following must be true.

(i) It has really crashed, in which case the failure detector guarantees that it will be eventually suspected by N_i, which will thus propose a new view.

(ii) It has installed a different view and thus it is not accepting the old master's proposals. In such case, the *exclusion* mechanism described in sect. 4.2.5 will also notify a failure of N_q, causing N_i or its successor to propose a different view excluding N_q.

If N_q installs $V_{\rm i}$, i.e. performs the $\texttt{commit}_q(V_{\rm i})$ event, but installs another view, $V_{\rm k}$, as immediate successor of $V_{\rm i}$, then one of the following is true.

(i) $p \in V_{\rm k}$, in which case

$$\texttt{prepare}_p(V_{\rm k}) \overset{*}{\prec} \texttt{commit}_q(V_{\rm k}),$$

so that N_p has entered a later view change which will eventually end with the installation of a view (see Proposition **4**); or

(ii) $p \notin V_{\rm k}$, so that the *exclusion mechanism* will eventually notify N_p about all nodes in $V_{\rm i}$ that are now excluding it from their views, in particular N_q. That will lead to a new view proposal led by the corresponding master of the subgroup where N_p belongs.

□

A.1.3 Liveness Properties

Membership Precission (Property GM.5)

Proposition 6. (Property GM.5) *Let S be a stable component. Then there is a committed view $V_{\rm f}$ such that $N \in V_f \quad \forall N \in S$, and V_f is installed as the last view by all $N \in S$.*

Proof. Let $N_i \in S$ and $N' \notin S$. Since S is a stable component, any N' is not connected to N_i. Assuming a reasonable behaviour of the system, the failure detector can behave as $\diamond\mathcal{P}_r$. Then it will eventually suspect from any such N', so that it will be excluded from the group. As there are no more messages coming from N' and eventually no delayed message will remain in the network channels, either, we may state that eventually all nodes which do not belong to S will be excluded from any view to be installed.

On the other hand, let us consider two nodes in the stable component, N_i, $N_j \in S$, and let us assume they have installed different views V_i and V_j, respectively. We may assume that V_i, $V_\mathrm{j} \subseteq S$, according to the discussion above. If $V_\mathrm{i} \cap V_\mathrm{j} \neq \emptyset$, i.e. they are not disjoint, then $\exists N_\mathrm{p} \in S$ such that $N_\mathrm{p} \in V_\mathrm{i} \cap V_\mathrm{j}$, so that we can establish an order between both views. Let us assume (without loss of generality) that $V_\mathrm{i} \overset{*}{<}_\mathrm{p} V_\mathrm{j}$. Then, if $N_i \in V_\mathrm{j}$, since $\texttt{commit}_j(V_\mathrm{j})$ happened, N_i has executed $\texttt{prepare}_i(V_\mathrm{j})$. As there are no more failures, it will eventually `commit` V_j (see proposition **4**). If, conversely, $N_i \notin V_\mathrm{j}$, then the exclusion mechanism ensures that N_i will eventually install a different view, such that it is disjoint with respect to V_j.

Then if N_i and N_j stay in disjoint views, since at least one of them is in minority, it will send **Join** messages probing for a group to join, and, being S stable, those messages will eventually reach the other group thus launching the *Partition Merging* protocol, and allowing N_i and N_j to install the same view.

Hence, eventually all nodes in S will install the same view V_f, not including any $N' \notin S$. Since all nodes in S remain correct and connected, no more failures will be ever suspected, so that V_f becomes the definitive view. □

A.1.4 Additional Uniform Majority Agreement

Besides the basic specification of the GMP from sect. 3.1.2, the HMS protocol also satisfies an additional property regarding the uniform maintenance of the majority history by the whole system.

Uniform Majority View Agreement (Property GM.6)

This Agreement property ensures that no confirmed majority views are missed by any participating node, if this is to take part in a later majority group.

In order to show how the HMS membership service satisfies this property, we start proving a few intermediate results.

Proposition 7. *The protocol of View History Matching discards only non* `committed` *views.*

Let N_A, N_B be two nodes undergoing a Partition Merging process, such that their majority histories are synchronised up to the last confirmed view, but hold different `prepared` majority views.[1] Let V_A and V_B be their respective `prepared` majority views. We take, without loss of generality, $V_A < V_B$. We will show that V_A can safely be discarded, i.e. it has never been `committed` by another node.

Proof. (Proposition **7**)

If one group is in majority (this may only be true for the group with the most up to date history), since V_A is not contained in the *majority history*, it has certainly been discarded by the majority group, and N_A must now do the same.

Therefore the only non-trivial situation is that when both N_A and N_B are in minority partitions, and they hold information about majority views that were in the process of being installed when they separated from the majority history.

If $N_B \in V_A$, given that N_B has not confirmed V_A (for it is not in its majority history), but has prepared a later view V_B, then the majority has already discarded V_A before preparing V_B.

On the other hand, if $N_A \in V_B$, V_A may also be safely discarded, since if it had been committed by the majority, the (V_A,`commit`) pair would have accompanied the preparation of any subsequent majority view, such as V_B in the corresponding **Setmem**.

So we are left with the case where $N_A \notin V_B$ and $N_B \notin V_A$. Nevertheless, since both V_A and V_B are majority views, they must contain common nodes. The only reason why V_A must not be discarded is that $\exists N_k \in V_A$ such that $\texttt{commit}_k(V_A)$ has been executed.

Let us assume that is the case. Then, since confirmation of majority views can only be ordered by the majority master, if this does not fail, commitment of V_A is communicated to all the surviving nodes before or while proposing a new view. So, any `prepare` event of a higher view will causally follow $\texttt{commit}(V_A)$ or $\texttt{upcommit}(V_A)$ in any node of the group, in particular in those which are common to V_A and V_B, which will then contain that confirmation in their majority history, regardless of how many majority views are proposed afterwards. Being not in V_A, N_B can only enter to a new proposed view V_B after joining the majority group. In such process, however, N_B receives the updates of the majority history, including confirmation of V_A before the **Setmem**(V_B) is sent. Therefore, V_A must be in N_B's majority history and N_A will also have confirmed it during the view matching process, what contradicts our hypothesis.

The only way for this not to happen is that the master of V_A fails before the `commit` order has reached all nodes in V_A. Nevertheless, in that case, the master changing protocol ensures that, if the majority character is conserved, as all surviving nodes had at least `prepared` V_A, the new master will also broadcast V_A confirmation with its

[1]Otherwise the problem of what to do with the oldest prepared view is trivial, since explicit information exists in the most up to date history.

new proposal, with the special label to `upcommit` it, so that it will equally enter the majority history with the corresponding `upcommit` event at each node.

If majority was lost between V_A and V_B, before V_B could be proposed by a **Setmem** message, an identical merging process should have taken place to match majority histories and recover majority, so that no previous V_A would remain pending. □

Proposition 8. (Previous result) *If* $\exists N_k$, V_j *such that* $\mathtt{commit}_k(V_j)$ occurs, *being* V_j *a majority view, then*

$$\mathtt{commit}_q(V_j) \overset{*}{\prec}_q \mathtt{prepare}_q(V_s), \tag{A.12}$$

or

$$\mathtt{upcommit}_q(V_j) \overset{*}{\prec}_q \mathtt{prepare}_q(V_s),$$

$\forall N_q$, V_s *such that* $V_s > V_j$ *is in majority and* $N_q \in V_s \cap V_j$.

Proof. (Proposition **8**) Let us assume that V_s is the majority proposal that immediately follows V_j, i.e. $V_j \overset{(1)}{<} V_s$. Since both views are in majority, the order relation of their identifiers is well-defined.

If neither a master change, nor a majority loss happens between V_j and V_s, the basic three-phase commit ensures that the proposition holds, since N_q will only perform $\mathtt{prepare}_q(V_s)$ after receiving a **Setmem** message also containing the commit instruction. If M is the Master of both V_j and V_s, applying relations (A.7) and (A.1),

$$\mathtt{commit}_M(V_j) \overset{*}{\prec} \mathtt{commit}_p(V_j),$$

and

$$\mathtt{commit}_M(V_j) \overset{*}{\prec}_M \mathtt{prepare}_M(V_s) \overset{*}{\prec}_M \mathit{send}_M(\textbf{Setmem}).$$

If N_q had not yet confirmed the commission of V_j, the **Setmem** message necessarily includes the confirm instruction and

$$\mathit{recv}_p(\textbf{Setmem}) \overset{*}{\prec}_q \mathtt{commit}_q(V_j) \overset{*}{\prec}_q \mathtt{prepare}_q(V_s).$$

If the majority Master fails after committing V_j, its substitute will collect information from the rest of nodes and if some node saw the explicit confirmation of V_j it will be confirmed and included in every *majority history.* Two situations may arise then.

- The resulting group may keep the majority condition. Then, since V_j was (at least) prepared in each node (and no later view was proposed afterwards), the new Master will confirm it, by including (V_j,commit) or (V_j,upcommit) in the **Setmem**(V_s) message, so that each V_s participant will confirm V_j before preparing the next view. If SM is the substitute Master, since V_j was committed by the old Master (M) and N_q did not see the confirmation (otherwise the proposition holds trivially), by applying (A.6) we get,

$$\texttt{prepare}_k(V_j) \overset{*}{\prec} \textit{Master Change} \overset{*}{\prec} \textit{send}_{SM}(\textbf{Setmem})$$

$$\overset{*}{\prec} \textit{recv}_q(\textbf{Setmem}) \overset{*}{\prec}_q \texttt{(up)commit}_q(V_j) \overset{*}{\prec}_q \texttt{prepare}_q(V_s).$$

- The resulting group may be in minority. Then, since V_j is the last prepared majority view, it is kept by every node. When the majority is to be recovered, this view will be confirmed. More formally, if N_q did not receive even the **Setmem**(V_j) but some other surviving node, N_r, did,

$$\texttt{prepare}_r(V_j) \overset{*}{\prec} \textit{Master Change} \overset{*}{\prec} \textit{send}_{SM}(\textbf{Setmem}) \supset (V_j, \text{prepare})$$

$$\overset{*}{\prec} \texttt{upprepare}_q(V_j) \overset{*}{\prec} \textit{send}(\textbf{Join/Joined/Update}) \supset V_j$$

$$\overset{*}{\prec} \textit{send}_{SM}(\textbf{Setmem}(V_s)) \supset (V_j, \text{upcommit})$$

$$\overset{*}{\prec} \textit{recv}_q(\textbf{Setmem}) \overset{*}{\prec}_q \texttt{upcommit}_q(V_j) \overset{*}{\prec}_q \texttt{prepare}_q(V_s).$$

  If N_q had already received the `prepare` instruction, the first line of the last inequality may be left out.

If the majority Master fails before locally committing V_j and the majority is lost due to this failure, but some of the participants had already received the **Setmem**(V_j), the result is the same as in the second case, so that a possibly committed view is never discarded by the majority.

If V_s does not immediately follow V_j but other majority views are proposed in between, but the majority is not lost, the same argument may be applied in each node to the first majority view, V', that is locally prepared after V_j, regardless of whether V' is committed or discarded afterwards. On the other hand, if the majority is lost and then recovered, proposition **7** ensures that no possibly committed view will be discarded. □

These propositions ensure that committed majority views are never dismissed, but enter the majority history of any node that takes part of a later view also in majority.

Proposition 9. (Property GM.6) *If* $\exists N_p \in V_k$ *such that* $\texttt{commit}_p(V_k)$ *is executed, then* $\forall N_q \in V_k$ *and* $\forall V_j > V_k$ *such that* N_q `commits` V_j,

$$\texttt{commit}_q(V_k) \overset{*}{\prec}_q \texttt{commit}_q(V_j),$$

or

$$\texttt{upcommit}_q(V_k) \overset{*}{\prec}_q \texttt{commit}_q(V_j),$$

as long as V_k *and* V_j *are both in majority.*

Proof. (Property GM.6)

We have $N_p \in V_k$ such that $\texttt{commit}_p(V_k)$ and $N_q \in V_k \cap V_j$ such that $\texttt{commit}_q(V_j)$ and $V_k < V_j$.

Since local `prepare` precedes the corresponding `commit`,

$$\texttt{prepare}_q(V_j) \overset{*}{\prec} \texttt{commit}_q(V_j),$$

the `prepare` event for V_j does happen in N_q. Then, applying proposition **8**, we must conclude that the older view V_k is included in the majority history of N_q, and

$$\texttt{(up)commit}_q(V_k) \overset{*}{\prec} \texttt{prepare}_q(V_j).$$

So the thesis trivially follows. □

A.2 HaloMS Protocol

The HaloMS membership service satisfies the formal properties specified in sect. 5.3.2. Here we sketch the proof for every property.

A.2.1 Safety Properties

Unicity (Property HM.1)

The unicity property is guaranteed by the token algorithm, since it solves the mutual exclusion problem, and thus ensures that clients will be added by only one core node at a time.

To avoid the possibility of an obsolete message launching a duplicate joining attempt, maybe even after the client is abandoning the group, any core node receiving a client's joining request asks for confirmation by means of a **JoinQst** message, before trying to add the corresponding proposal to the token.

Validity (Property HM.2)

This is trivially satisfied by the proposed algorithm, since a client can only include a core node in its *horizon* if it has received its identity as an argument in a **Horizon** message. Such message can only be issued by the client's representative after completing a token round within a single core view, Therefore every member of the view gets to know the client's identity and to include it in its *halo view*. The **Horizon** message thus contains only nodes that have processed the right token entry.

Consistency (Property HM.3)

The consistency property requires that two different core nodes do not disagree on the identifier of a given client.

Proof. Let us assume that the opposite happens, i.e. there exist two core nodes, N_r and N_s holding local entries in their respective *halo views* assigning a different identifier to the same client, C. Let us assume, without loss of generality, that N_r started the addition of C in the first place. Before adding an entry for a new client, a core node must process the token and analyse its own halo to check that the client is neither being added by another node, nor already confirmed as definitive. Therefore, addition and removal of C must have been completed by N_r before N_s receives the joining request. Otherwise, an entry for C is still present in the token, and N_s will dismiss the client's message. There are only two possibilities for this to happen.

- The client's request arrives after the post-definitive round has passed by N_s, so that it had already removed the local entry from its *halo view*. Then, N_s will ask the client for confirmation, and try to add it to the group with a new identifier when it gets the token back. But then the last token round has finished, thus eliminating any entry for C from all core members, so that N_r does not maintain a different identifier.

- The second possibility is that N_s was not part of the core view that granted the identifier ID_r for C, but has joined the group after the definitive confirmation of C. Then an obsolete join request from C may reach N_s, that has no information about C being already part of the halo. However, when receiving the client's request, N_r must contact C for a confirmation. Since C has already been granted an identifier, it will not confirm the request and will not be added by N_s.

□

A.2.2 Liveness Properties

Halo Liveness (Property HM.4)

If a client voluntarily disconnects, it must inform its representative. When the disconnection is due to a failure, no notification to the core group takes place. In that case, the satisfaction of this property depends on the behaviour of the representative's failure detector. We are assuming that the latter will eventually suspect the client. Thus the representative will start a **Remove** round and the identifier that was granted to the client will be finally invalidated.

Horizon Liveness (Property HM.5)

(i) The requirement that if a core member fails it is eventually eliminated from all clients' *horizon* is guaranteed by the recovery procedure after a reconfiguration of the core group. When a stable view is recovered, each original or inherited representative informs its clients about failed nodes, so that they are removed from *membership horizons*.

(ii) The eventual **Update** rounds guarantee the second *horizon* liveness requirement, namely that if a core member joins the group and both the client and the core node stay connected, the former will eventually include the latter in its *horizon*. When the client's representative decides to launch an updating round, the client will be added to the *halo view* of every core member in the group, and finally a new **Horizon** message to the client will notify it about the current group of servers that are aware of its presence.

A.3 MODUS Protocol

A.3.1 Consistent Membership (Property OD.1)

The property of consistent Membership requires that registered membership listeners receive the service of a basic group membership service satisfying the basic safety properties in sect. 3.1.2. MODUS preserves this property, since when a membership client registers at a given node, this enters the ACTIVE state and launches its local RMS, which in turn provides membership information with the four basic safety properties GM.1-GM.4.

There is a special consideration to be done here, nevertheless. Property GM.2, as already mentioned earlier in this appendix, refers to view oriented systems, where each event must occur in a certain view. Obviously in the context of an on demand service, we are assuming that there will be components or operation modes that are independent of membership information, whose events do not need the view context. The Initial View property, thus, only applies to operation modes that are *membership–aware*. For these situations, the property is guaranteed as long as the *membership–aware* operation starts by registering as a membership listener to MODUS.

A.3.2 Active Membership Precision (Property OD.2)

This property is basically the liveness property GM.5 of the fundamental GMP specification reduced to stable *active* components. Since the original enunciate of this property referred to stable physical components, it cannot be fulfilled by an On–Demand service that allows correct nodes not to take part of membership agreement. The only guarantee that can be required concerns only active members. For those, the property is trivially fulfilled, as it is granted by the RMS itself. In fact, for the underlying RMS, one may consider *connected* only those nodes whose instance of RMS is running, as the others will not take part of the protocol, neither send **Join** messages to enter the group. The connected nodes, then, will be provided with the basic property GM.5.

A.3.3 Eventual Stop (Property OD.3)

This is the fundamental property that guarantees that MODUS can stop the underlying RMS and thus reduce the message and energy consumption during periods of inactivity, when membership information is not required.

Proposition 10 ((Property OD.3)). If $\exists t_1$ such that $\forall t > t_1$ every possible membership client stays unregistered. Then $\exists t_2 \geq t_1$ such that $\forall t > t_2$, N_p stays inactive $\forall N_p$.

Proof. Let us consider a time $t > t_1$. Since after t_1 all clients stay disconnected, no node will be in the ACTIVE state, so that no **ServReq** message will be sent. We may choose a t long enough so that there is no such message in the system. Thus, those nodes that are in the UNAVAILABLE or in the AVAILABLE state will not connect their RMS from that point on. Then we are only concerned about nodes in the PASSIVE state.

Let us consider a node N_1 in such state. If N_1 holds an empty *activeNodes*, MODUS will stop its RMS, as required by this property.

If, conversely, $\exists N_2 \in activeNodes$, i.e. N_1 believes a second node to be ACTIVE, two cases may occur.

(i) N_2 is in the AVAILABLE state. In this case, its own RMS has already been stopped. Therefore it will be eventually excluded from N_1 view (because of the GM.5 property of the RMS protocol) and the latter will correspondingly remove it from its *activeNodes* local list, thus enabling its own stop.

(ii) N_2 is in the PASSIVE state. Then its own *activeNodes* must contain also some node. It is easy to see that if there is no cycle in the relationship *contained in the activeNodes of*, then all such nodes will eventually remove all the others from their own views and stop their RMS. The only non trivial case corresponds thus to a cycle. Let us first consider the simplest case,

$$N_1 \in N_2.activeNodes \quad \wedge \quad N_2 \in N_1.activeNodes.$$

In this situation, if N_1 and N_2 have been in common views since N_2 became PASSIVE,[2] then the **PassiveMsg** message from N_2 must eventually reach N_1 (as it is multisent until received unless N_1 is excluded from N_2's view, what did not occur). Therefore N_1 will remove N_2 from its *activeNodes* list, which will become empty, so that N_1 will stop its RMS.

If, on the contrary, N_2 did exclude N_1, then it would have also removed it from its own *activeNodes* table. If when rejoining, N_2 had received an **ActiveList** message from a third node, containing N_1 as ACTIVE, the corresponding updating **PassiveMsg** from N_1 would have followed the view change. In any case, the cycle would disappear, as N_2 would have an empty *activeNodes*, and both nodes would eventually stop their RMS.

It is easy to see that in case of a longer or more connected cycle, very similar reasonings can be applied to show that either members stay connected and eventually receive **PassiveMsg** messages from each other, or after partition they exclude disconnected members, so that the cycle eventually disappears.

□

[2]Notice this had to occur, since N_2 had to be ACTIVE to enter N_1.*activeNodes* in the first place, and now N_2 is PASSIVE.

Appendix B

Input/Output Automata. Formalisation of Algorithms

All the algorithms designed and analysed in this thesis were formalised as input output automata in order to reason about their properties. This appendix describes the scheme developed for supporting their easier implementation in Java. The last part of the appendix contains the formal specification of the presented algorithms in this formalism.

B.1 The Input/Output Automata Formalism

The Input/Output Automaton model (IOA) [64, 65] is one of the existing tools for modelling distributed systems. It is suited for the modelling of concurrently executing discrete event systems. Each system component is modelled as an Input/Output (I/O) automaton.

An I/O automaton A is a labelled state transition system. It consists of the following components.

- An *action signature*, $sig(A)$, which is a partition of the set of actions $acts(A)$ into three disjoint sets (*input*, or $in(A)$, *output*, or $out(A)$, and *internal*, or $int(A)$).
- A set of states, $states(A)$, possibly infinite.
- A nonempty subset of *start states*, $start(A) \subseteq states(A)$.
- A transition relation $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ such that for every state $s \in states(A)$ and action $\pi \in acts(A)$, there is a triplet $(s, \pi, s') \in steps(A)$.

An action π is said to be enabled in state s if there exists some triplet $(s, \pi, s') \in \mathit{steps}(A)$. IOA are input enabled, so that input actions are required to be enabled in all states. This means that the automaton cannot block its input. The transition relation of an automaton is described by giving a precondition and an effect (or postcondition) for each action (preconditions can be omitted if they are *true*). Therefore, action π will be enabled in every state s that fulfils its precondition.

A complex system can be modelled by composing several automata that model simpler system components. For a collection of automata, $\{A_i\}$, to be composable, they are required to be strongly compatible, i.e. their signatures must satisfy the following conditions.

1. $\mathit{out}(A_i) \cap \mathit{out}(A_j) = \emptyset$, $\forall i \neq j$,
2. $\mathit{int}(A_i) \cap \mathit{acts}(A_j) = \emptyset$, $\forall i \neq j$,
3. no action can be contained in infinitely many $\mathit{acts}(A_i)$.

Then the composition of the collection is another automaton, $A = \prod_i A_i$ defined as follows.

- $\mathit{sig}(A) = \prod_i \mathit{sig}(A_i)$ defined by
  - $\mathit{in}(A) = \cup_i \mathit{in}(A_i) - \cup_i \mathit{out}(A_i)$,
  - $\mathit{out}(A) = \cup_i \mathit{out}(A_i)$,
  - $\mathit{int}(A) = \cup_i \mathit{int}(A_i)$.
- $\mathit{states}(A) = \prod_i \mathit{states}(A_i)$, the Cartesian product of the state sets of every automaton, and being s_i the ith component of the state vector $\vec{s}$.
- $\mathit{start}(A) = \prod_i \mathit{start}(A_i)$.
- $\mathit{steps}(A)$ is the set of triplets $(\vec{s}, \pi, \vec{s'})$ such that, $\forall i$, if $\pi \in \mathit{acts}(A_i)$ then $(s_i, \pi, s'_i) \in \mathit{steps}(A_i)$, and if $\pi \notin \mathit{acts}(A_i)$ then $s_i = s'_i$.

When an automaton runs, it generates an execution, i.e. a sequence of alternating states and actions $s_0, \pi_1, s_1, \pi_2, \ldots$, such that s_0 is a start state and every triplet satisfies $(s_i, \pi_{i+1}, s_{i+1}) \in \mathit{steps}(A)$. A fair execution is such that each component is given the chance to execute its locally controlled actions. For a more formal definition of fairness, a partition $\mathit{part}(A)$ of each automaton's local actions (i.e. $\mathit{int} \cup \mathit{out}$) into equivalence classes is introduced to capture the structure of the modelled system. Each class should represent the set of actions controlled by an elemental component of the system. Formally, an execution is said to be fair if it satisfies the following condition.

1. If the excution is finite, then no action is enabled in the final state.
2. If the execution is infinite, then for each equivalence class C in $part(A)$, either the executions contains infinitely many events from C or it contains infinitely many states in which no action from C is enabled.

In the IOA model a problem is specified as set of allowable behaviours (i.e. sequences of input an output actions from an execution). An automaton is said to solve a problem if for all its fair executions, the corresponding sequence of external actions is contained in that set.

Different formalisms can be applied to model partially synchronous systems. For real-time systems, Lynch and Vaandrager defined the formalism of timed automata [102, 103, 74]. However, for the timing algorithms presented in this work, it is enough to consider the MMT model [74]. This is a simple variation of the I/O automaton model in which the fairness conditions are replaced by lower and upper time bounds. The timed MMT automaton is then an I/O automaton in which every equivalence class in $C \in part(A)$ is assigned a lower and an upper bound on time, $t_{inf}(C)$ and $t_{sup}(C)$. The automaton execution is timed, i.e. each action in the history is associated with a non-negative time value. These values are required to be non decreasing and to satisfy such bounds. This means that once C is enabled in a state,

- no action of C occurs before t_{inf}; and
- if time passes beyond $t_{sup}(C)$, in the interim either an action of C occurs, or C is disabled.

For the protocols described in this work we are mainly using timeouts. These are controlled by the periodic decreasing of dedicated counters. The actions that have as an effect such decreasing, then, have lower and upper bounds. For the rest of local actions, the lower bound can be reduced to 0. Therefore every automaton needs only two equivalence classes (or tasks).

B.2 The Implementation of IOA in Java

All our implementations have been done after the I/O automata specification of the corresponding algorithms. In this section, we present the main ideas in the Java implementation of the IOA model.

The scheme used for the implementation of our automata is an abstract Java class called `IOAutomaton`, shown in the UML diagram of fig. B.2. To imlement an automaton, a class must inherit `IOAutomaton` and implement the abstract method `checkState()`, which is assumed to check the state of the automaton for enabled actions and invoke one of them, if any. This class includes an inner thread, `Checker`, in charge of verifying which local actions may be enabled after any change of state. To ensure that the proper checking is done, the `actionFinished` method must be invoked after execution of any automaton action since the effect of any action may result in the enabling of others.

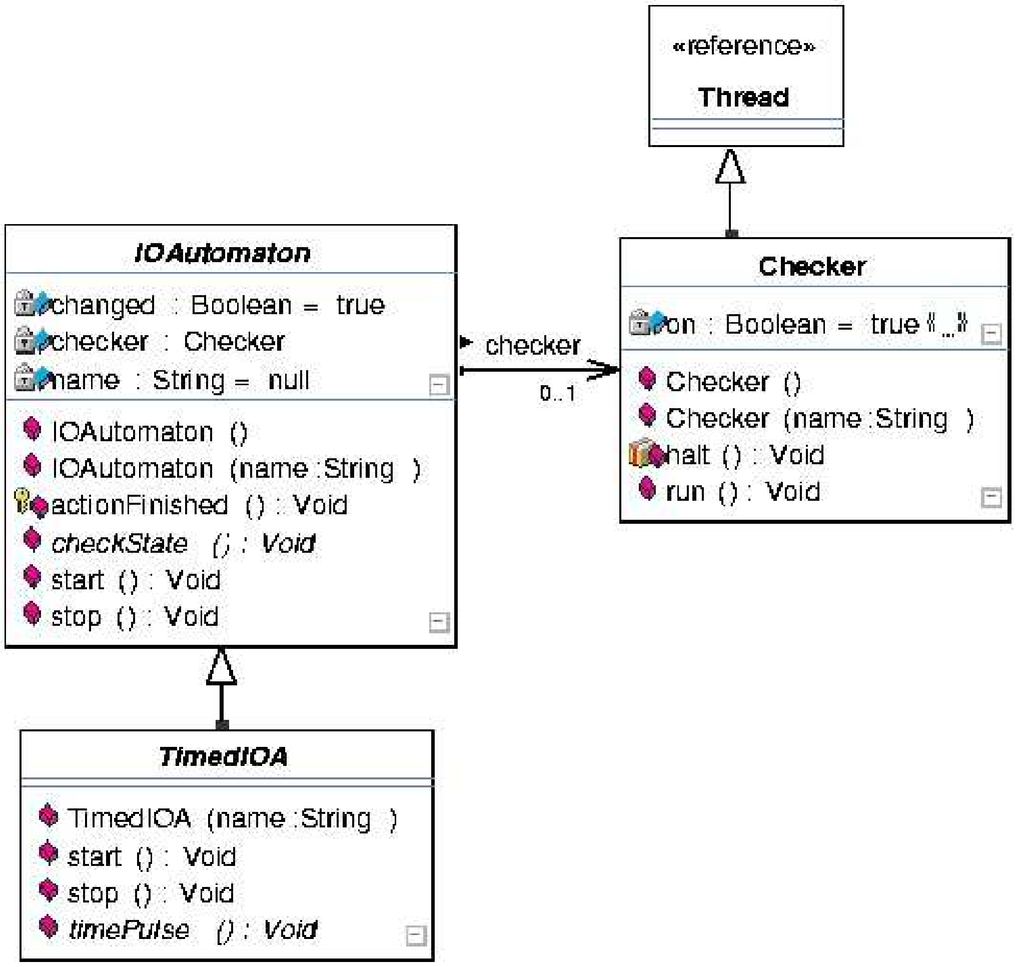


Figure B.1: Fundamental classes for the implementation of Input/Output Automata in the HAMS architecture.

For those components with timing restrictions we have extended the basic class `IOAutomaton` with the abstract class `TimedIOA`. This includes the abstract method `timePulse()`, that has to be implemented by any particular automaton with timeouts.

Such method is in charge of decreasing all the required counters maintained by the automaton state.

The TimedIOA class ensures that this action is periodically invoked, within the limits

All implemented TimedIOA use a basic time unit or pulse. Their counters and timeouts are then to be defined in terms of such basic time unit of the system. A TimeManager static entity is used to register all the created TimedIOA implementations and periodically invoke their timePulse() methods. For a more efficient handling, the registered timed automata are divided into bounded groups, and each of them is notified by a Timer thread. The main classes used for the implementation of timed automata are shown in B.2.

For all our automata we have developed a specific IOAData class which contains the state of the corresponding automaton. This class implements all the automaton actions, with proper synchronisation to ensure their atomicity. Besides, each automaton is realised as a particular class implementing IOAutomaton (or TimedIOA). In each such class, the checkState method retrieves the instant state of the automaton from the corresponding data class. Then it calculates a vector of booleans containing the status (enabled or not) of every local action, and randomly selects one of the enabled ones to be executed, and invokes actionFinished() after that is done. The implementation of each particular IOAutomaton must also include visible methods for input actions, and explicitly call actionFinished() after their execution.

Some of our algorithms are modelled as a composition of automata. In those cases the composition is achieved by an external class. Invoked by the external actions of each individual automaton, this class transforms such transitions into input actions for the concerned automata of the composition, ensuring also the proper synchronisation.

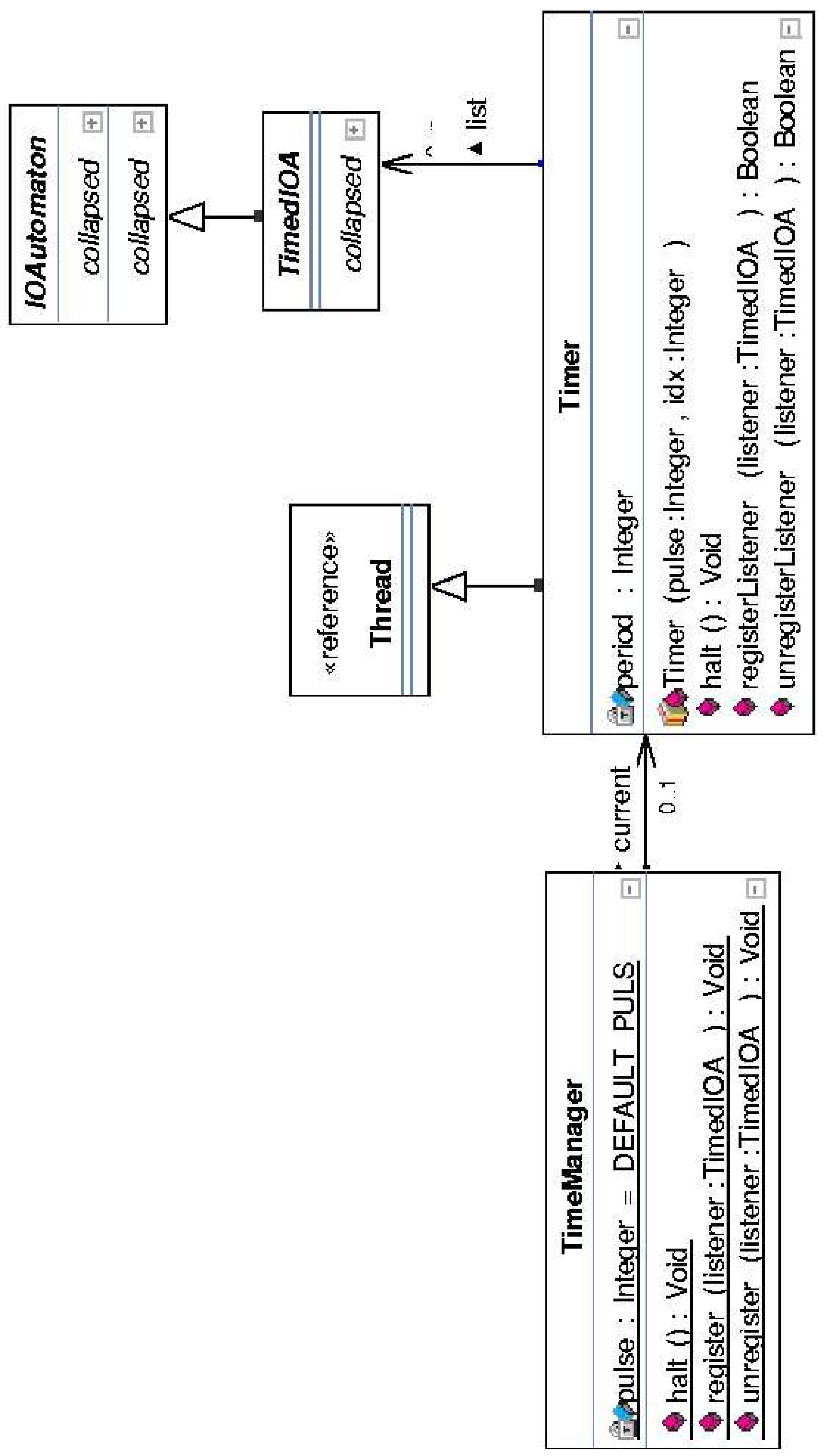

IOAutomaton
collapsed
collapsed
TimedIOA
collapsed
«reference»
Thread
list
Timer
period : Integer
Timer (pulse :Integer , idx :Integer)
halt () : Void
registerListener (listener :TimedIOA) : Boolean
unregisterListener (listener :TimedIOA) : Boolean
current
0..1
TimeManager
pulse : Integer = DEFAULT_PULS
halt () : Void
register (listener :TimedIOA) : Void
unregister (listener :TimedIOA) : Void

Figure B.2: Fundamental classes for the implementation of timed automata in the HAMS architecture.

B.3 Formalisation of the Membership Protocols

B.3.1 HMS

The HMS membership service is modelled as a two-folded composite automaton (Fig. B.3). Each component automaton takes charge of some specific task of the pro-

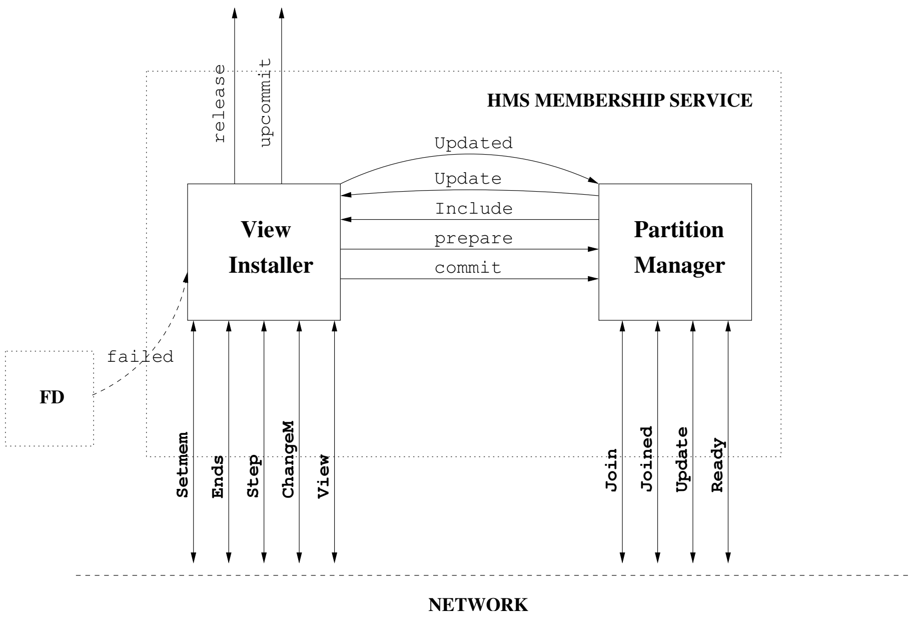


Figure B.3: Schematic view of the composition of two automata that defines the HMS membership service and its interaction with the local failure detector and the network.

tocol as follows.

- **View Installer** maintains the view lists (*strong* or *weak*) during view installation and performs the required operations during the *Master Changing* protocol, since this phase is closely dependent on the list contents. In the case of Master, it is also in charge of composing new views, from the information issued by the other component and the failure detector.

- **Partition Manager** maintains the *majority history* and takes charge of running the *Partition Merging* protocol. This involves both broadcasting and answering to **Join** requests. After a successful merging, an `Include` event must be issued, so that the **View Installer** component may prepare a new view including the freshly joined nodes.

The composition is schematically shown in the figure. Each automaton has two types of external actions. The first group corresponds to network messages exchanged with other nodes, which are denoted in the figure with bold typeface. The double arrows represent both the `send` (output) event and the `receive` (input) event corresponding to a given message. The second type corresponds to *local* events, in the sense that they are visible only for other local components. Some of them correspond to input events of the other automaton, also shown in the figure, whereas the rest will be visible for higher level users of the membership service.

Tables B.1 and B.2 contain the specification of each automaton's state and signature. There, every event related to the process of view installation has been made explicit.

Each automaton's transitions are described in deeper detail in tables B.3 and B.4. These tables collect the essential features associated with the different transitions, but some details (such as the management of view lists, the dismissing of obsolete or repeated messages and the multisending of some particular messages) have been omitted for brevity. They must, nevertheless be taken into account in a real implementation of the service.

View Installer Automaton (in node N_i)		
Action Signature		
Input	Output	Internal

`recv(msg)`
`Include(N)`
`Exclude(N)`
`Update(list)`
	`send(`**Setmem**`)`
	`send(`**Step(n)**`)`, $n \geq 1$
	`send(`**Ends(n)**`)`, $n \geq 0$
	`send(`**ChangeM**`)`
	`send(`**View**`)`
	`prepare(V)`
	`discard(V)`
	`commit(V)`
	`release(V)`
	`store(V)`
	`upcommit(V)`
	`Updated`
		`process(`**Setmem**`)`
		`process(`**Step(n)**`)`
		`process(`**Ends(n)**`)`
		`process(`**ChangeM**`)`
		`process(`**View**`)`
		`composeView`
		`timePulse`$_1$
		`stepTimeout`
		`chgTimeout`

State		

role $\in$ {MASTER, OTHER, SUBST} (initially MASTER);
view[V_{id}, *Master*, {*members*}] (initially [*zero*, N_i, {N_i}]);
strong →list of majority views (initially empty);
weak →list of minority views (initially empty);
nextV →Composition of next view (init. NULL);
acceptable[*Master*, *group*] (initially NULL)
step →integer value (initially 0);
step_done $\in$ {TRUE, FALSE} (initially FALSE);
prep, *disc*, *comm*, *rel* $\in$ {TRUE, FALSE} (init TRUE);
toPrep, *toComm* →Lists of views (initially NULL);
answers→list of nodes (initially NULL);
pending→list of received messages (initially NULL);
changeM $\in$ {TRUE, FALSE} (initially FALSE);
sendChgM $\in$ {TRUE, FALSE} (initially FALSE);
sendView $\in$ {TRUE, FALSE} (initially FALSE);
timers $= (timer_{step}, timer_{chg})$ →set of time counters to control timeouts;
timeouts →variables to indicate timeouts (initially FALSE).		

(Timed) Tasks		
{`timePulse`$_1$} with limits $t_{inf} = \text{PULSE}_1 - \Delta t$, $t_{sup} = \text{PULSE}_1 + \Delta t$ *local*(ViewInstaller)\{`timePulse`$_1$} with limits $t_{inf} = 0$, $t_{sup} = \text{PULSE}_1 + \Delta t$		

Table B.1: Signature and state variables of the View Installer automaton.

Partition Manager Automaton (in node N_i)		
Action Signature		
Input	Output	Internal

`recv(`**Join**`)`
`recv(`**Joined**`(`V_{id}`))`
`recv(`**Ready**`)`
`recv(`**Update**`)`
`commit(`V`)`
`prepare(`V`)`
`Updated`
`Exclude`
	`send(`**Join**`)`
	`send(`**Joined**`)`
	`send(`**Ready**`)`
	`send(`**Update**`)`
	`Include(`N`)`
	`Update(`*list*`)`
		`timePulse`$_2$

State		

role $\in$ {MASTER, OTHER} (initially MASTER);
status $\in$ {ACTIVE, INACTIVE} (initially ACTIVE);
minority $\in$ {TRUE, FALSE} (initially TRUE);
joining[...] (init. NULL);
updating[...] (init. NULL);
ready[V_{id}] (init. NULL);
update$\rightarrow$list of views for majority history (init. NULL);
joined$\rightarrow$ list of joined nodes (init. empty);
localU $\in$ {TRUE, FALSE} (init. FALSE);
timer $\rightarrow$ time counter to control timeouts;
prepM$\rightarrow$ used for recovering the majority (init. NULL).		

(Timed) Tasks		
{`timePulse`$_2$} with limits t_{inf} = PULSE$_2 - \Delta t$, t_{sup} = PULSE$_2 + \Delta t$ *local*(PartitionManager)\\{`timePulse`$_2$} with limits $t_{inf} = 0$, t_{sup} = PULSE$_2 + \Delta t$		

Table B.2: Signature and state variables of the Partition Manager automaton.

View Installer Automaton (in node N_i)

Input Transitions

```
Include($N$)
   Postcond.:
      if($role$ = MASTER) then
        $change \leftarrow$ TRUE
        $nextV \leftarrow nextV \cup N$
      fi

Exclude($N$)
   Postcond.:
      if($role$ = MASTER) then
        $change \leftarrow$ TRUE
        $nextV \leftarrow nextV \setminus N$
      else if($N$ is $Master$) then
        $role \leftarrow$ SUBST
        $nextV \leftarrow nextV \setminus N$
        $acceptable \leftarrow acceptable \setminus N$
        $changeM$, $sendChgM \leftarrow$ TRUE
      fi
```

```
recv($msg$)
   Postcond.:
      if($msg$ is acceptable) then
        $pending \leftarrow pending \cup msg$

Update($list$)
   Postcond.:
      if($step = 3$) AND
           ($step\_done$ = TRUE) then
        $step \leftarrow 4$
        $step\_done \leftarrow$ FALSE
        $toComm, toPrep \leftarrow list$
        $dis$, $prep$, $comm$, $rel \leftarrow$ FALSE
        if($role$ = OTHER) then
           $acceptable \leftarrow list$
        fi
      fi
```

Output Transitions

```
send(Setmem)
   Precond.:
      $role$ = MASTER
      $step = 0$
      $step\_done$ = TRUE
      $change$ = FALSE
   Postcond.:
      $toComm \leftarrow nextV$
      $answers \leftarrow nextV.members$
      $timer_{step}$ set
```

```
send(ChangeM)
   Precond.:
      $role$ = SUBST
      $step\_done$ = TRUE
      $changeM = sendChM$ = TRUE
   Postcond.:
      $sendChM \leftarrow$ FALSE
      $answers \leftarrow nextV.members$
      $nextV \leftarrow N_i$
      $timer_{chg}$ set
```

Table B.3: Transitions for the View Installer automaton.

View Installer Automaton (in node N_i)

```
send(Ends(n))
  Precond.:
    $role$ = OTHER
    $step = n$
    $step\_done$ = TRUE
    $changeM$ = FALSE
  Postcond.:
    $timer_{step}$ set
prepare($V$)
  Precond.:
    $step\_done$ = FALSE AND $step < 4$
    $toPrep = V$
    $prep$ = FALSE
    $comm = rel$ = TRUE
    ($disc$ = TRUE) OR
      ($\nexists V' < V$, prepared in $list$)
  Postcond.:
    $list \leftarrow list \cup$ (V,prepared)
    $toPrep \leftarrow$ NULL
    $prep$, $disc \leftarrow$ TRUE
    $step\_done \leftarrow$ TRUE
commit($V$)
  Precond.:
    $step\_done$ = FALSE AND $step < 4$
    $toComm.first = V$
    $comm$ = FALSE
    ($V$,prepared)$\in list$
    ($rel$ = TRUE) OR
      ($\nexists V' < V$, committed in $list$)
  Postcond.:
    $list \leftarrow list \cup$ (V,committed)
    $toComm \leftarrow toComm \setminus V$
    $comm$, $rel \leftarrow$ TRUE
    if(($step = 1$) then
      $step\_done \leftarrow$ TRUE
    fi
```

```
send(Step(n))
  Precond.:
    $role$ = MASTER
    $step = n$
    $step\_done$ = TRUE
    $change$ = FALSE
  Postcond.:
    $answers \leftarrow nextV.members$
    $timer_{step}$ set
discard($V$)
  Precond.:
    $step\_done$ = FALSE AND $step < 4$
    $prep = disc$ = FALSE
    ($V$,prepared)$\in list$
    (($rel = comm$ = TRUE) AND
        ($toPrep = V' > V$))
      OR ($toComm.first = V' > V$)
  Postcond.:
    $list \leftarrow list \setminus V$
    $disc \leftarrow$ TRUE
release($V$)
  Precond.:
    $step\_done$ = FALSE AND $step < 4$
    $rel$ = FALSE
    ($V$,committed)$\in list$
    ($toComm.first = V' > V$)
      OR ($step = 2$)
  Postcond.:
    $list \leftarrow list \cup$ ($V$,released)
    $rel \leftarrow$ TRUE
    if($step = 2$) then
      $step\_done \leftarrow$ TRUE
    fi
```

Table B.3: Transitions for the View Installer automaton.

View Installer Automaton (in node N_i)	

`send(`**View**`)`
Precond.:
$changeM = \text{TRUE}$
$step\_done = \text{TRUE}$
$sendView = \text{TRUE}$
Postcond.:
$sendView \leftarrow \text{FALSE}$
$timer_{chg}$ set
`Updated`
Precond.:
$((role = \text{MASTER})\text{AND}(step = 5))$
OR
$((role = \text{OTHER})\text{AND}(step = 4))$
$step\_done = \text{TRUE}$
Postcond.:
$step \leftarrow 3$
`store(`V`)`
Precond.:
$step\_done = \text{FALSE}$
$toComm.first = V \not\ni N_i$
Postcond.:
$toComm \leftarrow toComm \setminus V$
	`upcommit(`V`)`
	Precond.:
	$step\_done = \text{FALSE}$
	$toComm.first = V$
	$step = 4$ AND $comm = \text{FALSE}$
	Postcond.:
	if(V',`committed`)$\in strong$) AND
	($V' < V$) then
	set V' `released`
	fi
	$strong \leftarrow strong \cup (V,$`committed`$)$
	$toComm \leftarrow toComm \setminus V$
	if($toComm \neq \text{NULL}$) then
	$comm \leftarrow \text{FALSE}$
	else
	if($toPrep \neq \text{NULL}$) then
	$strong \leftarrow strong \cup (V,$`prepared`$)$
	$toPrep \leftarrow \text{NULL}$
	fi
	$step\_done \leftarrow \text{TRUE}$
	fi

Internal Transitions	

Table B.3: Transitions for the View Installer automaton.

View Installer Automaton (in node N_i)

```
process(Ends(n))
  Precond.:
    Ends(n) msg first in pending
    $step\_done =$ TRUE
  Postcond.:
    if($msg$ matches $acceptable$) then
      if($role$=MASTER)
            AND($n = step$) then
        $answers \leftarrow answers \setminus sender$
        if($answers$ = null) then
          $step \leftarrow step + 1$
          if($step < 5$) then
            $rel \leftarrow$ false
            $step\_done \leftarrow$ false
          fi
          if($step = 1$) AND
              ($change$=FALSE) then
            $toComm \leftarrow$ nextV
            $comm \leftarrow$ false
          fi
          $timer_{step}$ reset
        fi
      fi
    fi

process(ChangeM)
  Precond.:
    ChangeM msg first in pending
    $step\_done =$ TRUE
  Postcond.:
    if($msg$ matches $acceptable$) AND
          ($role \neq$MASTER) then
      if($role \neq$ SUBST) then
        $changeM \leftarrow$ TRUE
        $timer_{chg}$ started
      fi
      $sendView \leftarrow$ TRUE
      $acceptable \leftarrow acceptable \setminus Master$
      $nextV \leftarrow nextV \setminus Master$
      $timer_{step}$ reset
    fi
```

```
process(Step(n))
  Precond.:
    Step(n) msg first in pending
    $step\_done =$ TRUE
  Postcond.:
    if($msg$ matches $acceptable$)AND
        ($n = step + 1$) then
      $step \leftarrow n$
      $step\_done \leftarrow$ FALSE
      if($n = 2$) then
        $rel \leftarrow$ FALSE
      fi
      if($n = 1$) then
        $rel$, $comm \leftarrow$ FALSE
        $toComm \leftarrow$ nextV
      fi
      if($n = 4$) then
        $toComm$, $toPrep \leftarrow msg$
        $disc$, $prep$, $comm$, $rel \leftarrow$ FALSE
      fi
      $timer_{step}$ reset
    fi

process(View)
  Precond.:
    View msg first in pending
    $step\_done =$ TRUE
  Postcond.:
    if($role =$ SUBST) AND
        ($msg$ matches $acceptable$) then
      $answers \leftarrow answers \setminus sender$
      if($N_i$ results excluded) then
        $answers \leftarrow$ NULL
        $nextV \leftarrow$only $N_i$
      fi
      if($answers =$ NULL) then
        $role \leftarrow$ MASTER
        $changeM \leftarrow$ FALSE
        $change \leftarrow$ TRUE
        $timer_{chg}$ reset
      fi
    fi
```

Table B.3: Transitions for the View Installer automaton.

View Installer Automaton (in node N_i)	
`process(`**Setmem**`)` Precond.: **Setmem** *msg* first in *pending* *step_done* = TRUE Postcond.: if(*msg* matches *acceptable*) then *role* ← OTHER *step* ← 0 *step_done* ← FALSE if(*changeM* = TRUE) then *changeM* ← FALSE fi *toPrep* ← $msg.V_p$ *prep*, *disc* ← `false` *toComm* ← $msg.V_c$ if(*toComm* ≠ NULL) then *comm*, *rel* ← `false` fi *timers* are reset *acceptable* ← $[msg.sender, V_p.members]$ fi `timePulse`$_1$ Precond.: Postcond.: for *t* in *timers* $t \leftarrow t - 1$ if($\exists timer_j = 0$) then $timeout_j \leftarrow$ TRUE $timer_j$ reset fi	`composeView` Precond.: *role* = MASTER (*change* = TRUE) OR ($timeout_{step}$ = TRUE) *step_done* = TRUE Postcond.: if(*changeM* = TRUE) then *nextV*, *toComm* ← computed *changeM* ← FALSE else *comm*, *rel* ← TRUE fi if($timeout_{step}$ = TRUE) then $nextV \leftarrow View \setminus answers$ $timeout_{step} \leftarrow$ FALSE fi *toPrep* ← *nextV* *prep*, *disc* ← FALSE *step* ← 0 *step_done*, *change* ← FALSE

Table B.3: Transitions for the View Installer automaton.

View Installer Automaton (in node N_i)	
`timePulse`$_1$ Precond.: Postcond.: for t in $timers$ $t \leftarrow t - 1$ if($\exists timer_j = 0$) then $timeout_j \leftarrow$ TRUE $timer_j$ reset fi `stepTimeout` Precond.: $timeout_{step} =$ TRUE $role =$ OTHER Postcond.: $timeout_{step} \leftarrow$FALSE	`chgTimeout` Precond.: $timeout_{chg} =$ TRUE Postcond.: $timeout_{chg} \leftarrow$FALSE if($role =$ SUBST) then $role \leftarrow$ MASTER $changeM \leftarrow$ FALSE $change \leftarrow$ TRUE else if(N_i new subst) then $role \leftarrow$ SUBST $sendChM \leftarrow$ TRUE $nextV \leftarrow nextV \setminus oldSubst$ fi fi

Table B.3: Transitions for the View Installer automaton.

Partition Manager Automaton (in node N_i)

Input Transitions

```
recv(Join)
  Postcond.:
    if($role$ = MASTER) AND
        ($status$ = ACTIVE) then
      if(more updated history) then
        if(relative majority) then
          $joining \leftarrow$[sender.$group$,
            history difference,prepM]
        else
          $updating \leftarrow$ [sender,
            history difference,prepM]
        fi
      fi
    fi
recv(Update)
  Postcond.:
    if(($role$ = MASTER) AND
        ($status$ = ACTIVE)) AND
        ($minority$ = TRUE)) then
    $update,prepM \leftarrow msg.contents$
    $status \leftarrow$ INACTIVE
    fi
commit($V$)
  Postcond.:
    if($localU$ = FALSE) then
      $minority \leftarrow$ isMinority($V$)
      if($V.Master = N_i$) then
        $role \leftarrow$ MASTER
        $timer \leftarrow 0$
      else
        $role \leftarrow$ OTHER
        $timer$ reset
      fi
      $status \leftarrow$ ACTIVE
      $joined$, $joining \leftarrow$ NULL
    fi
```

```
recv(Joined($V_{id}$))
  Postcond.:
    if($status$ = ACTIVE) AND
        ($ready$ = NULL) AND
        ($msg$ acceptable) then
      $update,prepM \leftarrow msg.contents$
      $ready \leftarrow V_{id}$
      if($update \neq$ NULL) then
        $status \leftarrow$ INACTIVE
      fi
    fi
recv(Ready)
  Postcond.:
    if(($role$ = MASTER) AND
        ($status$ = ACTIVE)) then
    $joined \leftarrow joined \cup msg$.sender
    fi
Updated
  Postcond.:
    $localU \leftarrow$ FALSE
    $status \leftarrow$ ACTIVE
    $joining \leftarrow$ NULL
prepare($V$)
  Postcond.:
    if($localU$ = FALSE) then
      $status \leftarrow$ INACTIVE
      $joined \leftarrow$ NULL
      $timer$ reset
      if($V$ majority) then
        $prepM \leftarrow V$
      fi
    fi
```

Table B.4: Transitions for the Partition Manager automaton.

Partition Manager Automaton (in node N_i)	
`Exclude(`N_j`)` Postcond.: $status \leftarrow$ INACTIVE $joined \leftarrow$ NULL $timer$ reset	
Output Transitions	

`send(`**Join**`)`
Precond.:
$role =$ MASTER
$status =$ ACTIVE
$minority =$ TRUE
$timer = 0$
Postcond.:
$timer \leftarrow$ reset
`send(`**Update**`)`
Precond.:
$role =$ MASTER
$status =$ ACTIVE
$updating \neq$ NULL
Postcond.:
$updating \leftarrow$ NULL
`Include(`N_j`)`
Precond.:
$status =$ ACTIVE
$role =$ MASTER
$joined = N_j$
Postcond.:
$joined \leftarrow$ NULL
	`send(`**Joined**`)`
	Precond.:
	$role =$ MASTER
	$status =$ ACTIVE
	$joining \neq$ NULL
	Postcond.:
	$joining \leftarrow$ NULL
	`send(`**Ready**`)`
	Precond.:
	$ready \neq$ NULL
	$update =$ NULL
	$status =$ ACTIVE
	Postcond.:
	$ready \leftarrow$ NULL
	`Update(`$list$`)`
	Precond.:
	$update = list$
	Postcond.:
	$update \leftarrow$ NULL
	$localU \leftarrow$ TRUE

Internal Transitions	

Table B.4: Transitions for the Partition Manager automaton.

Partition Manager Automaton (in node N_i)	
`timePulse`$_2$ Precond.: Postcond.: if($timer$ set) then $timer \leftarrow timer - 1$ fi	

Table B.4: Transitions for the Partition Manager automaton.

B.3.2 HaloMS

The HaloMS membership service runs differently in server and client nodes. In

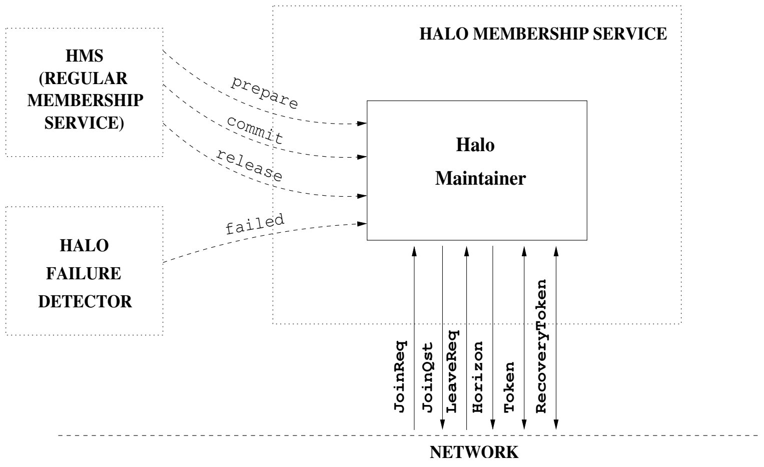


Figure B.4: Automaton that models the HaloMS membership component running in each core node, and its interaction with the Regular Membership Service and the network.

the former, it is modelled as an automaton, Halo Maintainer, that interacts with the Regular Membership Service component, the network and the local failure detector in charge of monitoring client connections. The automaton signature is schematically shown in fig. B.4 for the particular case of HMS. In client nodes, on the contrary, the whole membership service consists only of the HaloMS component, so that this interacts only with the network and the local failure detection component (see fig. B.5).

The signature of the automaton for the HaloMS membership component in server

Halo Maintainer Automaton (in server node N_i)		
Action Signature		
Input	Output	Internal

`recv(`**Token**`)`
`recv(`**Recovery**`)`
`recv(`**JoinReq**`)`
`recv(`**LeaveReq**`)`
`prepare(`V`)`
`commit(`V`)`
`release(`V`)`
`failed()`
	`send(`**JoinQst**`)`
	`send(`**Token**`)`
	`send(`**Recovery**`)`
	`send(`**Horizon**`)`
		`processToken`
		`checkInit`
		`recoverToken`

State		
status $\in$ {ON, RECOVERY, OFF} (initially OFF); *mode* $\in$ {WAIT, COMP, SEND} (initially WAIT); *role* $\in$ {INIT, OTHER} (initially OTHER); $View$ (initially NULL); $haloView$ (initially EMPTY); *token* $\in$ {TRUE, FALSE} (initially FALSE); *joining*$\rightarrow$list of (*client*, ref) pairs (init. NULL); *toAdd*$\rightarrow$list of clients to add (init. NULL); *toRemove*$\rightarrow$list of clients to be removed (init. NULL); *toClient* $\rightarrow$**Horizon** messages to be sent to clients (init. NULL); *toQst* $\rightarrow$**JoinQst** messages to be sent to clients (init. NULL); $nextToken$ (initially NULL); $recoveryToken$ (initially NULL); $lastToken$ (initially NULL); *initiator*$\rightarrow$node that heads the ring (init. NULL).		

Table B.5: Signature and state variables of the automaton that describes the behaviour of the HaloMS Service on the core side.

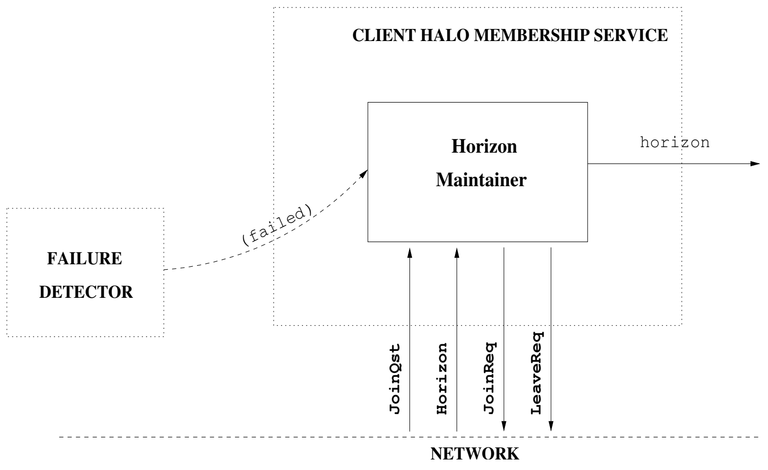


Figure B.5: Automaton that models the HaloMS membership component running in each client node, interacting with the network.

nodes, named Halo Maintainer, is schematically shown in table B.5.

The main aspects of the transitions are detailed in table B.6, which shows the dependence of the HaloMS component on the information coming from the regular membership service, in the case of HMS or a similar service that provides the events `prepare`, `commit` and `release`. We have chosen this type of support because the variety of membership events notified by HMS allows the HaloMS component to be simpler and to react more efficiently to changes in the core group. If the underlying RMS did only provide notification about installed views, the automata should be slightly modified, so that the token stop and recovery were launched by view events, as already discussed in the text.

The description of the automaton is schematically shown in the following tables. Details about token processing and updating of the *halo view*, have been deliberately left out for the sake of clarity. Such actions take place at every node whenever it is in turn of token processing, i.e. atomically with the `processToken` action. The specific details to update the local *halo view* with the incoming token have already been fully discussed in the text.

Halo Maintainer Automaton (in server node N_i)

Input Transitions

```
recv(Token)
  Postcond.:
    $token \leftarrow$ TRUE
    $lastToken \leftarrow msg$
    $mode \leftarrow$ COMP
    if($status$ = RECOVERY) then
      $status \leftarrow$ ON
      $role \leftarrow$ OTHER
      $initiator \leftarrow msg.T_{id}$
    fi
recv(Recovery)
  Postcond.:
    if($status$ = OFF) then
      $status \leftarrow$ RECOVERY
      $role \leftarrow$ OTHER
      $mode \leftarrow$ SEND
      $token \leftarrow$ FALSE
      $nextToken \leftarrow$ NULL
    fi
    $recoveryToken \leftarrow msg$
recv(LeaveReq)
  Postcond.:
    if($status$ = ON) then
      if(client $\in haloView$) then
        $toRemove \leftarrow toRemove \cup$ client
      fi
    fi
prepare($V$)
  Postcond.:
    $status \leftarrow$ OFF
    if($V$ is minority) then
      reset everything
    fi

recv(JoinReq)
  Postcond.:
    if($status$ = ON) then
      if(client $\notin haloView$) then
        if(client $\notin joining$) then
          if($msg.ref$ =NULL) then
            $joining \leftarrow joining$
            $\cup$(client, new reference)
            $toQst \leftarrow toQst$
              $\cup$ JoinQst(client,ref)
          fi
        else if($msg.ref = ref$ for $client$
          in $joining$) then
          $joining \leftarrow joining \setminus$ client
          $toAdd \leftarrow toAdd \cup$ client
        fi
      fi
    fi
commit($V$)
  Postcond.:
    $View \leftarrow V$
release($V$)
  Postcond.:
    if(($status$ = OFF) AND
        ($V$ is majority)) then
      $status \leftarrow$ RECOVERY
      $role \leftarrow$ OTHER
      $mode \leftarrow$ SEND
      $token \leftarrow$ FALSE
      $nextToken \leftarrow$ NULL
      $recoveryToken \leftarrow$ NULL
    fi
```

Output Transitions

Table B.6: Schematic transitions for the Halo Maintainer automaton.

Halo Maintainer Automaton (in server node N_i)

```
send(Token)
  Precond.:
    $status$ = ON
    $token$ = TRUE
    $mode$ = SEND
    $toClient$ = NULL
  Postcond.:
    $token \leftarrow$ FALSE
    $nextToken \leftarrow$ NULL
    $mode \leftarrow$ WAIT
send(Horizon)
  Precond.:
    $status$ = ON
    $token$ = TRUE
    $mode$ = SEND
    $toClient \neq$ NULL
  Postcond.:
    $toClient \leftarrow toClient \setminus msg$
```

```
send(Recovery)
  Precond.:
    $status$ = RECOVERY
    $role$ = OTHER
    $mode$ = SEND
  Postcond.:
    if($recoveryToken$ = NULL) then
      $mode \leftarrow$ WAIT
    else
      $mode \leftarrow$ COMP
    fi
    $token \leftarrow$ FALSE
    $nextToken \leftarrow$ NULL
send(JoinQst)
  Precond.:
    $status$ = ON
    $toQst \neq$ NULL
  Postcond.:
    $toQst \leftarrow toQst \setminus msg$
```

Internal Transitions

```
processToken
  Precond.:
    $status$ = ON
    $token$ = TRUE
    $mode$ = COMP
  Postcond.:
    $haloView \leftarrow$ updated
    $toAdd \leftarrow$ NULL
    $toRemove \leftarrow$ NULL
    $toClient \leftarrow$ computed
    $nextToken \leftarrow$ updated
    $mode \leftarrow$ SEND
```

```
checkInit
  Precond.:
    $status$ = RECOVERY
    $mode$ = COMP
    $recoveryToken \neq$ NULL
  Postcond.:
    $role \leftarrow$ computed
    $recoveryToken \leftarrow$ NULL
    $mode \leftarrow$ WAIT
    if($role$ = INIT) then
      $initiator \leftarrow N_i$
    fi
```

Table B.6: Schematic transitions for the Halo Maintainer automaton.

Halo Maintainer Automaton (in server node N_i)	
`recoverToken` Precond.: $status =$ RECOVERY $role =$ INIT Postcond.: $nextToken \leftarrow$ computed $toClient \leftarrow$ computed $token \leftarrow$ TRUE $status \leftarrow$ ON $mode \leftarrow$ SEND	

Table B.6: Schematic transitions for the Halo Maintainer automaton.

The automaton corresponding to the client side of the HaloMS service, called Horizon Maintainer, is described in tables B.7 and B.8. The first one shows the signature of the automaton, depicted also in fig. B.5, and the latter details the main aspects of the transitions. As before, details regarding resending of **JoinReq** if **JoinQst** or **Horizon** are not received within a reasonable time are omitted to keep the description concise. Regarding the local failure detector module in the client side, we assume for it only the capability to detect failed links. It is possible that a client is capable to contact some core nodes but not others, and therefore it can detect the latter have failed. In order to fulfil the required properties of the HaloMS service, nevertheless, a client should not be allowed to exclude a certain core node, as the responsibility to decide about representativeness and inclusion/exclusion of clients relies on the core group. Therefore the reaction to `failed` input transitions in the client side is optional, and we do not include it explicitly in this schematic description. In practice, this reaction can be to reduce the quality of the corresponding link in the local *horizon*, and communicate a new `horizon` event, so that the applications using this information can decide consequently on their operations involving that link.

Horizon Maintainer Automaton (in client node C_i)		
Action Signature		
Input	Output	Internal

`recv(`**JoinQst**`)`
`recv(`**Horizon**`)`
`joinGroup`
`leaveGroup`
	`send(`**JoinReq**`)`
	`send(`**JoinReq(**ref**)**`)`
	`send(`**LeaveReq**`)`
	`horizon`	

State		

status $\in$ {ON, OFF} (initially OFF);
connected $\in$ {TRUE, FALSE} (initially FALSE);
tryJoin $\in$ {TRUE, FALSE} (initially FALSE);
tryLeave $\in$ {TRUE, FALSE} (initially FALSE);
newHorizon $\in$ {TRUE, FALSE} (initially FALSE);
representative server node (initially NULL);
$reference$ integer (init NULL);
horizon list of (*core*,*link*) (initially NULL);		

Table B.7: Signature and state variables of the automaton that describes the behaviour of the HaloMS Automaton on the client side.

Horizon Maintainer Automaton (in client node C_i)

Input Transitions

```
recv(**JoinQst**)
  Postcond.:
    $representative \leftarrow msg.sender$
    $reference \leftarrow msg$
    $tryJoin \leftarrow$ FALSE
recv(**Horizon**)
  Postcond.:
    $representative \leftarrow msg$
    $horizon \leftarrow msg$
    $connected \leftarrow$ TRUE
    $newHorizon \leftarrow$ TRUE
```

```
joinGroup
  Postcond.:
    $status \leftarrow$ ON
    $tryJoin \leftarrow$ TRUE
    $connected \leftarrow$ FALSE
leaveGroup
  Postcond.:
    $status \leftarrow$ OFF
    $tryLeave \leftarrow$ TRUE
```

Output Transitions

```
send(**JoinReq**)
  Precond.:
    $status =$ ON
    $tryJoin =$ TRUE
    $connected =$ FALSE
    $representative =$ NULL
  Postcond.:
    $tryJoin \leftarrow$FALSE
send(**JoinReq**($ref$))
  Precond.:
    $status =$ ON
    $tryJoin =$ TRUE
    $connected =$ FALSE
    $representative \neq$ NULL
    $reference = ref$
  Postcond.:
    $tryJoin \leftarrow$FALSE
```

```
send(**LeaveReq**)
  Precond.:
    $status =$ OFF
    $tryLeave =$ TRUE
    $connected =$ TRUE
  Postcond.:
    $tryLeave \leftarrow$FALSE
horizon
  Precond.:
    $connected =$ TRUE
    $newHorizon =$ TRUE
  Postcond.:
    $newHorizon \leftarrow$FALSE
    if($horizon =$NULL) then
      $connected =$FALSE
    fi
```

Table B.8: Schematic transitions for the Horizon Maintainer automaton.

B.3.3 MODUS

The MODUS protocol has been modelled as a single automaton, MODUSAuto, whose signature is shown in fig. B.6. Besides the network, it interacts with the Regular Membership Service, filtering its notifications (`MbshipEvent) actions`, and with any user application (or general component) capable to register membership listeners.

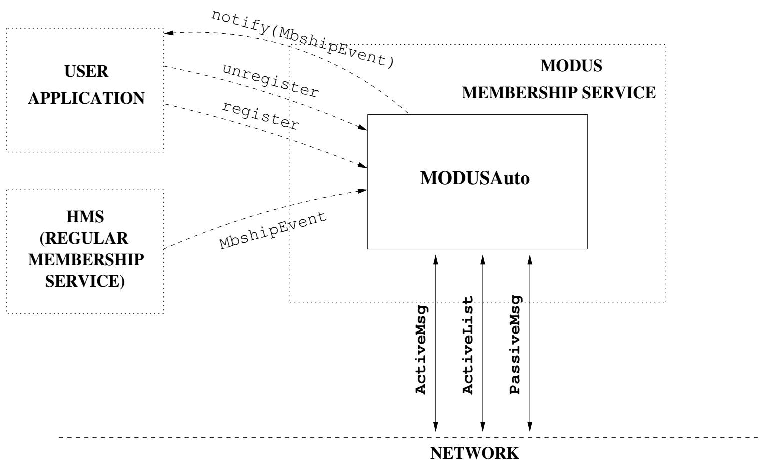


Figure B.6: Automaton that models the MODUS membership service, and its interaction with the Regular Membership Service, user applications and network.

The schematic signature of the automaton is shown in table B.9. The table shows the processing of **ActiveMsg** and **PassiveMsg** messages, taking into account that they may arrive disordered with respect to view changes and other messages. Thus, the MODUS algorithm delays the processing of advanced messages until the proper view has been installed. In case of partition merging, one representative of each partition send an **ActiveList** with the portion of the *activeNodes* list before the merging. Every survivor from the same partition waits for this message and, if necessary, sends a new message to update its own status, as described in sect. 6.4.3. This process is shown explicitly in the automaton. The variables *waiting*, *delayed* and *toHandle* take charge of this.

Nodes from other partitions must also process the **ActiveList** before those individual messages updating the state of particular nodes in response to that message. In order to keep the clarity of this description we have deliberately excluded the details about how **ActiveMsg** and **PassiveMsg** messages from disjoint partitions are ordered with respect to their corresponding **ActiveList**. This can be achieved by including a reference to the precedent view in the since the **ActiveList** and in the individual **ActiveMsg** and **PassiveMsg** that follow it, and proceeding in a similar way as the one described above.

MODUS Automaton (in node N_i)		
Action Signature		
Input	Output	Internal
`register`(*client*) `unregister`(*client*) `recv`(**ServReq**) `recv`(**ActiveMsg**(V)) `recv`(**ActiveList**(V,*nodes*)) `recv`(**PassiveMsg**(V)) `MbshipEvent`	`broadcast`(**ServReq**) `mcast`(**ActiveMsg**) `mcast`(**PassiveMsg**) `notify`(*MbshipEvent*)	`timePulse`$_3$ `switchoff` `handle`(**ActiveList**) `handle`(**ActiveMsg**) `handle`(**PassiveMsg**)
State		
$status \in \{$ON, OFF$\}$ (initially OFF); $role \in \{$ACTIVE, PASSIVE$\}$ (initially PASSIVE); *view* membership information (initially NULL); $activeNodes$ →list of active nodes (initially NULL); $localClients$ →list of registered clients (initially NULL); $toSend$ →list of messages to be sent (init. NULL); $waiting \in \{$TRUE, FALSE$\}$ (initially FALSE); $toHandle$ →list of messages to be processed (init. NULL); $delayed$ →list of messages that have to wait to be processed (init. NULL); $toNotify$ →list of membership events to be notified (init. NULL); $timer$ →time counter to control timeouts;		
(Timed) Tasks		
$\{$`timePulse`$_3\}$ with limits $t_{inf} = \text{PULSE}_3 - \Delta t$, $t_{sup} = \text{PULSE}_3 + \Delta t$ $local(\text{MODUS}) \setminus \{$`timePulse`$_3\}$ with limits $t_{inf} = 0$, $t_{sup} = \text{PULSE}_3 + \Delta t$		

Table B.9: Signature and state variables of the automaton that describes the behaviour of the MODUS Service.

MODUSAuto Automaton (in node N_i)

Input Transitions

```
register(client)
  Postcond.:
    $localClients \leftarrow localClients \cup client$
    if($status$=OFF) then
      $status \leftarrow$ ON
      $toSend \leftarrow toSend \cup$ ServReq
    else if($role$ =PASSIVE) then
      $toSend \leftarrow toSend \cup$ ActiveMsg($N_i$)
    fi
    $role \leftarrow$ ACTIVE
    $activeNodes \leftarrow activeNodes \cup N_i$
unregister(client)
  Postcond.:
    $localClients \leftarrow localClients \setminus client$
    if($localClients$ is empty) then
      $role \leftarrow$ PASSIVE
      $activeNodes \leftarrow activeNodes \setminus N_i$
      $toSend \leftarrow toSend \cup$ PassiveMsg($N_i$)
    fi
recv(ActiveMsg(V))
  Postcond.:
    if($status$ = ON) then
      if($V = view$) AND
         ($waiting$ =FALSE) then
         $toHandle \leftarrow toHandle \cup msg$
      else
         $delayed \leftarrow delayed \cup msg$
      fi
    fi
```

```
MbshipEvent(evt)
  Postcond.:
    $toNotify \leftarrow toNotify \cup evt$
    if($evt$ is MBSHIP_VIEW($V$)) then
      $lastview \leftarrow V$
      $view \leftarrow evt$
      $toHandle \leftarrow$ NULL
      move from $delayed$ to $toHandle$
         ActiveList($V$), ActiveMsg($V$)
         from joined partition
      $activeNodes \leftarrow activeNodes \setminus$ failed
      if($view \setminus lastview \neq \emptyset$) then
         if($N_i$ first from $lastview$) then
            $toSend \leftarrow$ ActiveList($V$,
                 $activeNodes$)
         else
            $waiting \leftarrow$TRUE
         fi
      fi
    fi
recv(PassiveMsg(V))
  Postcond.:
    if($status$ = ON) then
      if($V = view$) AND
         ($waiting$ =FALSE) then
         $toHandle \leftarrow toHandle \cup msg$
      else
         $delayed \leftarrow delayed \cup msg$
      fi
    fi
```

Table B.10: Schematic transitions for the MODUS automaton.

MODUSAuto Automaton (in node N_i)

```
recv(ActiveList(V,nodes))
  Postcond.:
    if($status$ = ON) then
      if($V = view$) then
        $toHandle \leftarrow toHandle \cup msg$
      else
        $delayed \leftarrow delayed \cup msg$
      fi
    fi
```

```
recv(ServReq)
  Postcond.:
    if($status$=OFF) then
      $status \leftarrow$ ON
      $timer \leftarrow$ reset
    fi
```

Output Transitions

```
broadcast(ServReq)
  Precond.:
    $status$=ON
    $msg \in toSend$
  Postcond.:
    $toSend \leftarrow toSend \setminus msg$
mcast(ActiveList)
  Precond.:
    $status$=ON
    $msg \in toSend$
  Postcond.:
    $toSend \leftarrow toSend \setminus msg$

mcast(ActiveMsg)
  Precond.:
    $status$=ON
    $msg \in toSend$
  Postcond.:
    $toSend \leftarrow toSend \setminus msg$
```

```
mcast(PassiveMsg)
  Precond.:
    $status$=ON
    $msg \in toSend$
  Postcond.:
    $toSend \leftarrow toSend \setminus msg$
notify(MbshipEvent)
  Precond.:
    $status$=ON
    $MbshipEvent \in toNotify$
    $localClients \neq \emptyset$
  Postcond.:
    $toNotify \leftarrow toNotify \setminus$
      $MbshipEvent$
```

Internal Transitions

Table B.10: Schematic transitions for the MODUS automaton.

MODUSAuto Automaton (in node N_i)

```
handle(ActiveList($view$,$nodes$))
  Precond.:
    $status$=ON
    $msg \in toHandle$
  Postcond.:
    $toHandle \leftarrow toHandle \setminus msg$
    $activeNodes \leftarrow activeNodes \cup nodes$
    if($sender$ = first from $lastview$) then
      $activeNodes \leftarrow activeNodes \setminus$
        ($lastview \setminus nodes$)
      $waiting$ ←FALSE
      if($role$ disagrees with $activeNodes$)
        then $toSend \leftarrow toSend \cup$
            PassiveMsg OR ActiveMsg
      fi
    fi
    move ActiveMsg($view$),
      PassiveMsg($view$) msgs
      from $delayed$ to $toHandle$
    if($activeNodes \neq \emptyset$) then
      $timer$ ←-1
    fi

timePulse₃
  Precond.:
  Postcond.:
    if($timer$ set) then
      $timer \leftarrow timer - 1$
    fi

switchoff
  Precond.:
    $status$=ON
    $activeNodes = \emptyset$
    $toSend = toHandle$ =NULL
    $timer \leq 0$
  Postcond.:
    $status$ ← OFF
    $delayed$ ← NULL

handle(PassiveMsg($view$))
  Precond.:
    $msg \in toHandle$
  Postcond.:
    $toHandle \leftarrow toHandle \setminus msg$
    $activeNodes \leftarrow activeNodes \setminus$
        $msg.sender$
    if($activeNodes = \emptyset$) then
      $timer$ reset
    fi

handle(ActiveMsg($view$))
  Precond.:
    $msg \in toHandle$
  Postcond.:
    $toHandle \leftarrow toHandle \setminus msg$
    $activeNodes \leftarrow activeNodes \cup$
        $msg.sender$
    $timer$ ←-1
```

Table B.10: Schematic transitions for the MODUS automaton.