Group Communication based on Standard Interfaces

Matthias Wiesmann[⋆] matthias.wiesmann@epfl.ch

Xavier Défago[†,‡] defago@jaist.ac.jp

André Schiper[⋆] andre.schiper@epfl.ch

[⋆] Distributed Systems Lab — Information & Communication Institute
Swiss Federal Institute of Technology in Lausanne (EPFL)

[†] Japan Advanced Institute of Science and Technology (JAIST)

[‡] "Information and Systems," PRESTO, Japan Science and Technology Corporation

Abstract

While group communication system have been proposed for some time, they are still not used much in actual systems. We believe that one reason for this is the lack of standardisation of group communication system interfaces. The paper proposes an architecture, using the standard decomposition into services, were services are based on standard interfaces: both interactions between services and interactions with the application use existing, open standards. A decomposition of the group communication into services is presented, along with a description of applicable standards. As an example, a group membership service based on the LDAP standard is discussed.

1. Introduction

Group communication systems, which provide *one-to-many* communication primitives with various semantics, have been an active area of research for more than a decade. The notion of process groups was proposed initially in the context of the V System [6], and later extended by the Isis system [3] to the context of fault-tolerance. Since then, the development of group communication has attracted a vast community of researchers. By providing high-level communication primitives with well defined properties [13], group communication systems make it easier to develop distributed fault-tolerant applications. Yet, despite tremendous advances in research and the existence of numerous prototypes (e.g., [23, 20, 31, 22, 9, 1, 2, 21]), group communication usage stays confined to small niches and to academic prototypes. While initial group communication systems have been monolithic [3], which may explain that they were not widely adopted, recent projects have proposed modular architectures [21, 16]. While such systems are more flexible and customizable to the user needs, this has not led to a significant increase in the use of the group communication technology.

However, there are recent examples of successful new communication technologies. A good example is the so-called *message oriented middleware* technology (MOM). MOMs offer abstractions for sending data asynchronously, either to a single recipient or to multiple recipients. Nowadays, MOMs are considered an integral part of the enterprise computing infrastructure. MOMs, but also the success of the World Wide Web, have shown that common standards are essential for the acceptance of a distributed system technology: standard interfaces promote interoperability, which is vital for the success of any group communication technology.

Common standards is probably what is missing in the context of group communication. We believe that in order to become accepted and used in a significant way, group communication systems need to adapt to the general network environment and to the interfaces applications might expect. Group communication systems usually assume that applications rely solely on the abstractions of group communication and are designed with a particular group communication framework in mind. This is, of course, not realistic. While designing more efficient or more flexible toolkits is an important research area, what is also needed is a more *usable* toolkit: we need standardized group communication interfaces. A standardised interface for group communication services could also avoid that other communities re-invent the wheel.

What standards do we want for group communication? We argue in the paper that there is probably no need to define new standards. Modern applications interact already with many standard network services. This can include low-level services like DNS or middleware infrastructures like message queues or object brokers. System configuration is increasingly done using standards interfaces like LDAP [32] and SNMP [4] and application level interaction are done using standardised MOMs like JMS. Why not integrate the group communication system with standard infrastructures?

Research to add group communication properties to MOMs has just started [26, 19] and questions on how applications can use them are still open. In this paper we propose a group communication architecture that emphasises interoperability and support for standard interfaces. The rest of the paper is structured as follows. Section 2 presents the design goals for a generic group communication framework. Section 3 describes the different technologies and standards that can be used for group communication interfaces. Section 4 illustrates the approach we propose to a specific group communication service, namely the *membership service*. Finally, Section 5 concludes the paper.

2. Design Goals and Architecture

In this section we describe the architecture of a group communication system that relies on standards. Our goal is to propose a general framework for interactions between group communication components, not to propose yet another group communication toolkit.

We assume a framework where each aspect of the group communication functionality is handled by one service. This granularity fits our goals (see Section 2.2) and matches proposals found in the literature (Section 2.1). We first present related work. Then we state the goal of such an approach. Finally, we describe an architecture that fulfils those goals.

2.1. Related Work

The approach of decomposing the group communication system into services is not new. Different services have been proposed for the design of group communication systems: failure detection [30, 10, 5, 15], group membership [18] and consensus [12]. All the proposed services have roughly the granularity described in this section.

While modular approaches are also considered in micro-protocol architectures [16, 21], the granularity is not the same: first, micro-protocols stacks tend to be built up from many small elements; second, the coupling is different. Micro-protocols are meant to be assembled and have little meaning in isolation. Services, on the other hand, should be usable as standalone infrastructure elements and offer meaningful interfaces to the users.

Another difference is the assembly philosophy: in micro-protocol architectures, users can add their own micro-protocols, as long as they are implemented using the framework's objects, threading and event models. Services, should simply provide standard interfaces and can be implemented in any way. Both approaches are in fact complementary. Services can be implemented using micro-protocols.

2.2. Design Goals

The architecture proposed in this paper is based on the following four design goals: *easy substitution, autonomy, self-containment,* and *standard.* As explained below, these goals are not always entirely independent.

Goal 1 (Easy Substitution) *A service implementation is easily replaced by another implementation providing the same function.*

For instance, consider S_α and S_β, two implementations of an arbitrary service S. To satisfy the goal, it should be easy to replace S_α by S_β in any given system. This is regardless of the fact that S_α and S_β might have different characteristics, e.g., scalability, performance, or even different semantics. To do so, S_α and S_β must necessarily have a common interface, but need not share anything beyond that.

Goal 2 (Autonomy) *A service retains its meaning when used in isolation from other services.*

While the different services are meant to be assembled into a complete group communication system, they should be usable and useful on their own. Among other things, each service should be accessible directly using the appropriate interface. Consequently, when applications do not need high-level properties, they are allowed to forgo the use of higher-level services and rely directly on the lower-level ones. An important implication of this goal for the architecture is that the system cannot rely on some “glue component” to link all services together. Indeed, the existence of this “glue component” would simply result in the definition of a new *de facto* middleware platform, thus completely defeating the purpose of the approach.

Goal 3 (Self-Containment) *Existing service implementations are easily integrated into the framework.*

Many existing products are related to group communication in a way or another: e.g., middleware platforms, administrative tools, database systems.[1] Obviously, reimplementing the same functionality within a group communication system hardly makes sense. Hence, we want to be able to integrate existing products into the framework, such as for instance message queue middleware platforms. To some extent, this goal can be seen as an extension of Goal 1 (Easy Substitution). Beside, Self-Containment does have an impact on applicable standards (Goal 4).

Goal 4 (Standard) *Services export their functionality through standard interfaces and protocols.*

In order to simplify both use and deployment, we require every service to adhere exclusively to standard interfaces, whenever one exists. The requirement applies to all services, regardless of how they are expected to be used (i.e., by applications or by other services). Note that, to a certain extent, this is already a way to implement Goal 1 (Easy Substitution) and Goal 3 (Self-Containment).

Currently, the specification of Fault-Tolerant CORBA (FT-CORBA) is the only standard definition that is directly related to group communication. Unfortunately, the FT-CORBA interface is strongly tied to the CORBA middleware specification. Yet, there are many standards in related domains that would be perfectly applicable for the design of a group communication system.

[1]Database systems often embed some of the functionalities usually provided by middleware platforms, in order to let clients connect to the database.

2.3. Architecture

In this section, we describe the general architecture of our proposed framework, to address the goals described in Section 2.2.

2.3.1 Overview

We consider that the system consists of a collection of mid-size components, each of which provides a given service. Services export a standard interface and use a collection of other interfaces for their implementation.

Given two implementations S_α and S_β of the same service S, S_α and S_β must necessarily export the same upper interfaces, as defined by S. However, the implementations may vary in their requirements, and thus S_α and S_β may or may not rely on the same lower interfaces. Nevertheless, we can say that two implementations S_α and S_β belong to the same class if they rely on the same set of lower interfaces in addition to the upper ones. In that case, S_α and S_β can always be used in place of another without requiring any structural change to the rest of the system.

The additional requirement that actually makes the originality of the approach is that all interfaces must be based on existing standards.

2.3.2 Group Communication Architecture

Based on these principles, Figure 1 shows a simplified architecture of a group communication system, as it could appear to a single node. All services provide standard interfaces and their implementation rely on the standard interfaces of the lower services (or those provided by the operating system). Each node contains a complete group communication system, with an instance of each service.

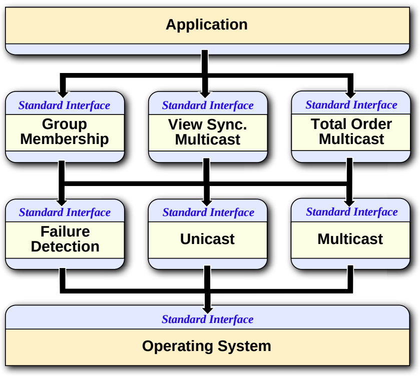


Figure 1. Overall Structure

This architecture decomposes group communication functionality in the following services:

Failure Detection: The failure detection service is responsible for collecting information about the status of the different processes, i.e., whether they have failed or not. Many distributed algorithms, in particular *agreement protocols*, rely on failure detectors or similar functionality.

Unicast: The unicast service is responsible for reliably transmitting messages from one process to another.

Multicast: The multicast service is responsible for reliably transmitting messages from one process to a set of processes.

Group Membership: The group membership service is responsible for maintaining information about groups and the processes that are member of those groups. This information is usually replicated on the different nodes of the system in a consistent manner [7].

View-Synchronous: The view-synchronous multicast service is responsible for multicasting (see above) messages with the additional guarantee that all processes member of a view deliver the same set of messages in that view (unless they are removed from the next view) [7].

Total Order Multicast: The total order multicast service is responsible for multicasting messages in such a way that all destination processes deliver the message in the same total order [13].

2.3.3 Discussion

Given the goals presented in Section 2.2 and the architecture sketched in Section 2.3, we can now explain how some specific details of the architecture relate to the mentioned goals.

The mid-size granularity of services is necessary to ensure that services are autonomous (Goal 2). On the one hand, if the decomposition into services is too coarse-grained, the system will be monolithic and leave little or no room for reuse. On the other hand, if the decomposition is too fine-grained, most of the services will have no utility beyond the context in which they have been developed. Besides, the burden of an excessively large number of standard interfaces for trivial tasks would almost certainly result in a major performance hog.

Autonomy (Goal 2) is ensured because each service exports its own interface and a tight coupling between different service implementations is discouraged. Similarly, easy substitution (Goal 1) is supported by the separation between interface, configuration, and implementation.

The preference for standard interfaces (Goal 4), contributes to address several issues. First of all, relying on existing standards ensures a steeper learning curve as many programmers are expected to be already familiar with APIs. Second, a system can be put together more quickly when the implementation of many parts actually consist of available products. It results that actual development is confined to services which must satisfy stronger semantics than usual, or simply whose performance must be strongly optimized.

3. Technology

The architecture described in Section 2.3 assumes that, for every service considered, there exist a possible match among standard interface specifications. Although the match might not always be perfect, we argue that an acceptable specification exists in almost every case.

In this section, we discuss different existing standards and see how they can be used for building our group communication architecture. We evaluate the adequacy of each standard interface based on the following three criteria:

Fitness: *Fitness* quantifies the adequacy of a standard with respect to our purpose. In other words, a given standard might be a perfect match or might require some adaptation (that is, not to say tweaks).

Acceptance: *Acceptance* measures the extent to which a standard interface is generally accepted or widespread, that is, whether it is actually used for many applications or not.

Openness: *Openness* evaluates how open a standard is. Certain standards are linked to a given software provider or bound to a specific programming language, whereas others originate from some standard-making body (i.e, IETF, ISO, ANSI). The latter are obviously better suited to our purpose.

Below we present standards according to their respective semantic functionality. For instance, there are many variations of broadcast semantics, but it is fitting for them to share a common interface. Doing so promotes easy substitution (Goal 1). The semantic classes that we consider in this paper are as follows: messaging, failure detection, and group membership. Other related semantic classes, including atomic commitment, remote invocation, or application-level replication, are not discussed in this paper.

It is important to remember that we consider standard *interfaces*. Indeed, several standards do not only specify interfaces, but also implementations. We however only consider interfaces and simply ignore whatever part of the standard relates to the implementation. As most of the standards described below have been designed outside the scope of group communication systems community, they lack the strong semantics considered in this context. Because of this, the definition of some standards has to be extended appropriately, so that they provide the proper guarantees (e.g., total order of delivery for total order multicast; see Sect. 2.3.2).

3.1. Messaging Technologies

Services in this category include the reliable unicast and broadcast services, but also advanced multicasts, such as causal, view-synchronous, or total order multicast. While unicast and multicasts are semantically different operations, having the same interfaces simplifies the design. In fact most standards try to offer similar interfaces for unicasts and multicasts (e.g., UDP datagrams and IP-multicast; see below).

3.1.1 TCP/IP, UDP/IP, IP-multicast

TCP/IP, UDP/IP, and IP-multicast are the basic and established standards for interprocess communications. The specification of all IP-based protocols is handled by the IETF.

TCP/IP [25] is a connection-oriented unicast protocol with message retransmission and flow control. UDP/IP [24] is a best-effort (aka, unreliable) message-oriented unicast protocol. IP-multicast is a message-oriented unreliable multicast protocol. UDP/IP and IP-multicast share the same interface. IP-multicast groups are simply represented by a special kind of destination address.

While very widely used, the TCP/IP interface is stream-oriented, and as such ill-suited for messaging (group communication systems tend to rely on messages). Also TCP/IP semantics implies FIFO semantics, which are often not needed by the application. The UDP/IP interface is message-oriented but has limited message length.

Both TCP and UDP are usually implemented inside the operating system. This means that in order to extend those interfaces, we would either need to change the networking stack of operating systems or use overlay networks. The first solution is clearly not practical, and the second implies that messages will be relayed by an operating system that does not understand the extended semantics. For instance if we implemented a reliable broadcast with an UDP interface using an overlay, the OS might still decide to drop messages (it is allowed to discard UDP messages).

Another issue is that code that uses the UDP interface usually includes functionality to handle its unreliable nature. UDP packets can be lost, duplicated, or delivered out of order. Because of this, adopting the UDP interface would have little benefit.

3.1.2 BEEP and APEX

BEEP and APEX are relatively recent protocols. The goal of BEEP [28] is to build a very simple common infrastructure for network protocols. BEEP is designed to be used on top of IP protocols. APEX [29] is a messaging protocol built on top of BEEP. According to its specification, APEX provides an extensible, asynchronous message relaying service for application-layer programs. APEX supports both unicast and multicast operations.

Because APEX is an application-level protocol with support for extensions, new implementations and advanced semantics can be used. For instance, APEX offers best-effort guarantees by default, but stronger guarantees can be implemented on top of that.

APEX is a very good standard for messaging infrastructure. Both APEX and BEEP are maintained by the IETF and are designed to be both lightweight and language-independent. The main drawback of those standards is that they are fairly recent and not widely adopted yet.

3.1.3 Message Queues

Message queues, often also called Message Oriented Middlewares (MOMs) offer an asynchronous messaging service. MOMs are widely used in enterprise settings, and many commercial products offer some MOM implementation. A message-queue interface is very interesting for high-level primitives that will be used by applications. Those applications often already connect to message queue interfaces to send messages. In fact research on group communication is increasingly targeting MOMs [19].

The Java message service specification (JMS) [14] is one existing standard for message-queue services. JMS is part of the Java 2 platform enterprise edition (J2EE) specification framework defined by Sun Microsystems and implemented by several vendors. JMS specifies both unicast and multicast interactions, respectively based on a message-queue model and a publish/subscribe model. The JMS specification accepts arbitrary information tags for messages, so additional semantic information can be attached, such as sequence numbers or view identifiers. This means that JMS is also a good standard for messaging interfaces, especially with respect to application-level interfaces.

3.1.4 Conclusion

APEX and JMS are two compelling candidates. APEX is lightweight, and better suited for low-level services. It is an open, language-independent standard. JMS is better accepted so far, but is tied to the Java platform. The most reasonable approach would be to use APEX as a core protocol, and add a JMS adapter for Java applications. A similar approach is used by the Joram project [19].

3.2. Technologies for Failure Detection

In this section, we describe services that are adapted to failure detection. Failure detection services offer a mean for processes to know about the status of other processes. Failure detection is usually done using *heartbeat* or *are-you-alive* messages, albeit more advanced techniques also exist (see [15] for a brief survey). Standard interfaces need to address mainly two issues. The first one is the standard used for accessing failure detection information and configuring the failure detection service. The second issue relates to the protocol used for sending *heartbeat* or *are-you-alive* messages. Failure detection as a service is increasingly considered in the literature [30, 10, 5, 15].

3.2.1 SNMP

The SNMP standard (Simple Network Management Protocol) [4] is widely accepted for network management purposes. Routers, switches, and network management tools all rely on the SNMP standard. Nowadays, many network-attached devices are configured and monitored using SNMP. This includes mission-critical servers, but also devices such as networked printers. The SNMP standard defines two interaction styles: a synchronous request-response mode, and an asynchronous notification system. In the SNMP lingo, notification messages are called traps. The SNMP standard is an open standard managed by the IETF.

SNMP is already used for failure monitoring, so using it for failure detection seems natural [27]. The request-response interaction style is suited for configuring the failure detection service. Meanwhile, the trap mechanism can be used for status messages. Typical trap messages are single packet UDP messages, and thus consume fairly little bandwidth.

3.2.2 CMIP

The CMIP standard (Common Management Information Protocol) is a protocol with more advanced features than SNMP. CMIP has seen little acceptance because of its heavyweight, and the fact that it is built to run on OSI networking stacks. CMIP is mostly used in the telecommunication devices area. However, since it does not support the IP protocol very well, CMIP is not a good candidate.

3.2.3 Conclusion

SNMP is the obvious choice for a failure detection service. Not only can the SNMP standard be used to implement a failure service detection service, but such a service could also collaborate with other SNMP devices to actually improve the detection of failure. Network elements such as routers or switches usually implement an SNMP interface themselves. So they can be programmed to send SNMP traps to signal relevant events, such as link failures or link overloads. Making good use of such information could greatly improve the accuracy of failure detectors. An implementation of failure detection using SNMP is described by [27].

3.3. Technologies for Group Membership

Technologies related to group membership (GM) are usually interfaces to access and manipulate directory information. At the core, a group is simply a set of processes and this information can be manipulated using directory access interfaces. In this section, we are only concerned about interfaces used to manipulate the group membership information, and not about guarantees (e.g, consistency) or actual mechanisms.

3.3.1 LDAP

The Lightweight Directory Access Protocol (LDAP) [32] is a standard defined by the IETF for accessing directory information. As such, it is well-suited for handling group membership information. LDAP is a widely accepted standard and is used increasingly to store system configuration information and administrative data. LDAP supports complex queries, and servers can asynchronously send multiple responses to one single request. In addition, LDAP supports federated repositories. The LDAP protocol is often used to access white page services, but also to read system configuration information. Among other things, LDAP is used as a core protocol for the Grid information protocol (GRIP), a foundation component of the Globus toolkit [8] (the protocol is responsible for managing all the configuration information for the Grid).

3.3.2 SNMP

The SNMP standard can handle very simple directories in the form of tables and hence could be used as an interface to a group membership service. The main problem with SNMP with respect to directory handling is its inability to process complex queries. Consequently, complex queries must be

Communication Component	Related Domain	Applicable Standards
Messaging	Networking, Message Queues	APEX, JMS
Failure Detection	Network Management	SNMP
Group Membership	Directory Access	LDAP, JNDI

Table 1. Applicable Technology

decomposed in a sequence of simple request/responses. Besides, the SNMP standard is unable to handle nested tables or aliases. Unfortunately, it turns out that nested tables would be necessary for a group membership service that supports hierarchical groups. Likewise, support for aliases would be required by federated group membership services.

3.3.3 JNDI

The Java Naming and Directory interface (JNDI) is a generic interface for accessing directory information from the Java language. The JNDI interface is part of the J2EE platform. The main advantage of JNDI is that it offers an abstract interface to directory information services. For instance, JNDI can be used to access LDAP servers.

3.3.4 Conclusion

Similar to messaging interfaces (Section 3.1), the most reasonable solution is to rely on the IETF standard (LDAP), as this standard is open and language-independent. For Java applications, a JNDI wrapper could also be provided easily. This would pose no problem, as JNDI supports LDAP-based implementations already. The specification of a group membership service using the LDAP standard is discussed further in Section 4.

3.4. Overview

Table 1 summarizes the different group communication components, the related domains and the applicable standards. Basically, there is at least one IETF-based standard relevant to each of the basic interfaces (i.e., APEX for messaging, SNMP for failure detection, and LDAP for group membership). Those standards impose a low overhead, as they rely on sockets for communication and impose no marshalling, which is an important performance bottleneck [11]. Additionally, services that will be accessed often by the application can additionally provide J2EE-compatible interfaces: JMS and JNDI.

Figure 2 details the architecture presented in Figure 1, using the actual standards. This shows that low-level services are in fact needed to implement other services. For instance, APEX is built top of BEEP, and the failure detection service relies on the functionality of SNMP. BEEP and SNMP are in turn implemented using IP-level interfaces.

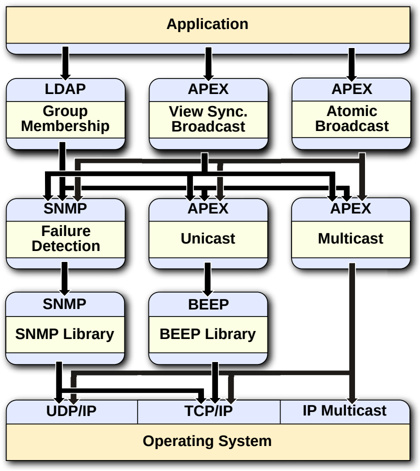


Figure 2. Overall Structure with standards

4. Illustration: Standard Interfaces for the Group Membership Service

In this section we show how standard interfaces can be used for the group membership service. As already mentioned, building low-level services with standard interfaces is a natural approach. We now show that this approach also works for higher level services, and illustrate this with the group membership service.

Most group communication systems maintain information about groups and their membership in a form or another. Based on this observation, Keidar et al. [18] have recently advocated that the group membership functionality should be provided as a service. Various definitions of group membership have already been proposed, with different semantics. In addition, the definition of the group membership of sometimes closely related or even dependent on the delivery of multicast messages (e.g., view synchronous broadcast). Because of this, the group communication system requires detailed information about the members of each group.

4.1. Group Membership Service

We now describe how the LDAP interface can be mapped to provide the functionality of a group membership service. In this paper, the term *group membership* is used with its broadest meaning. In particular, we do not assume or require strong semantics, such as consistent membership information among participating processes, or synchronization of message delivery with membership changes.[2]

A group G is a set of processes $\{p_1^G, \ldots, p_n^G\}$. Processes can be added to and removed from a group. A process p can be a member of several groups. Figure 3 illustrates this. There are two groups G_a and G_b, group G_a contains processes p_1, p_2, group G_b contains processes p_2, p_3, p_4. Each process p_i has a unique identifier. Additionally p_i has some

[2]There is no fundamental impossibility to supporting stronger semantics. However, this would make the presentation more complex with no significant additional insight.

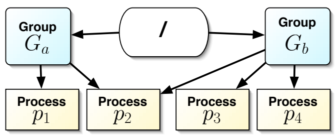


Figure 3. GM Structure

attached attributes that describe the process, for instance a transport layer address. The group membership service M manages all the groups $\{G_1, \dots, G_m\}$. Each node s_i contains an instance M_i of the group membership service. We assume that there is a weak coupling between the different instances of the group membership service, that is, in the absence of membership changes, the information held by instances is eventually consistent.

A minimal group membership service should implement the following operations: **Register** adds a given process to a given group. **Unregister** removes a given process from a given group. **Member of** tests whether a given process is member of a given group. **List members** returns the list of all members of a given group. **List groups** returns the list of all groups of which a given process is member. **Get info** accesses the information and properties of a given process.

In addition, we also consider the possibility that groups might be nested (i.e., a group is member of another group). To support this, the definition above is enlarged in such a way that group members can be not only process, but also other groups.

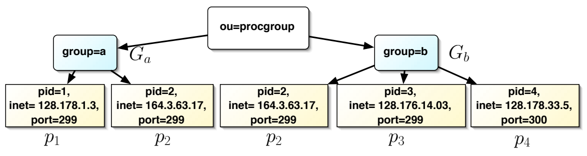
4.2. LDAP

Since LDAP is sometimes used to read system configuration information, it is well-suited to access and manipulate group membership information. By using this standard protocol, we ensure that external programs can interact directly with the system. For instance, a system administrator could use an LDAP client as a simple way to remove a process from a group.

4.3. Basic Group Membership based on LDAP

Applications and other group communication services access the membership on the local node. While the LDAP protocol can be used remotely this will not be used for the basic service. Information associated to a process include the node address of the process and the port number of the process.

The hierarchical structure described in Figure 3 can be modelled in the LDAP interface quite easily. We define processes as leaf objects with a certain number of attributes and groups as namespaces. The group attribute holds the name of the group a process is a member of. The pid attribute identifies the process within this group. The pid attribute serves as relative distinguished names (RDP), that uniquely identify an object within a namespace [17]. The transport layer address is represented by additional attributes. For instance, in the case of TCP/IP addresses, this results into two attributes: inet holds the IP address of the process and port holds the port number.



ou=procgroup
group=a
G_a
group=b
G_b
pid=1, inet= 128.178.1.3, port=299
pid=2, inet= 164.3.63.17, port=299
pid=2, inet= 164.3.63.17, port=299
pid=3, inet= 128.176.14.03, port=299
pid=4, inet= 128.178.33.5, port=300
p_1
p_2
p_2
p_3
p_4

Figure 4. LDAP Structure for basic GM

Figure 4 shows the LDAP structure corresponding to the situation described in Figure 3. All information queries can be done using the LDAP syntax, as illustrated by the following examples: To add process p_2 to group B (Register): insert(group=B, pid=2)[3]. To remove process p_2 from group B (Unregister): delete(group=B, pid=2). To test if process p_2 is member of group B (Member of): search(group=B, pid=2). To list all processes that are member of group B (List members): search(group=B, pid=*). To list all groups of which process p_2 is a member (List groups): search(group=*, pid=2). To obtain the properties of process p_2 w.r.t. group B: (Get info): search(group=B, pid=2).

5. Conclusion

In this paper, we have presented a general group communication architecture based on standard interfaces. We have discussed existing standards related to group communication sub-services. As a case study, we have presented an LDAP interface for a group membership service. This example shows that standards can be leveraged to offer a simple interface to group communication services and offer interesting features.

Our goal is not to propose a new standard, but rather to leverage existing standards. As the World Wide Web has shown, standardised networking interfaces are of primary importance for distributed applications. Among other things, increased interoperability means better acceptance of group communication systems.

Research on interactions between group communication system and the outside world is still at an early stage. More work is definitely needed to figure out how group communication and other infrastructure services can interoperate. We think that such an integration would contribute to a wider use of the very powerful group communication technology.

References

[1] A. Baratloo, P. E. Chung, Y. H. Huang, S. Rangarajan, and S. Yajnik. Filterfresh: Hot replication of java rmi server objects. In *Proceedings of the* 4^{th} *Conference on Object Oriented Technologies and Systems (COOTS)*, pages 59–63, Santa Fe, New Mexico, USA, 1998. USENIX.

[2] K. Birman, R. Constable, M. Hayden, C. Kreitz, O. Rodeh, R. van Renesse, and W. Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *Proceedings of the DARPA Information Survivability Conference & Exposition (DISCEX '00)*, Hilton Head, South Carolina USA, 2000.

[3] For simplicity the transport layer address attributes are omitted.

[3] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the* 11^{th} *ACM Symposium on OS Principles*, pages 123–138, Austin, TX, USA, 1987. ACM SIGOPS, ACM.

[4] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol (SNMP). RFC 1157, Internet Engineering Task Force (IETF), may 1990.

[5] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(2):561–580, may 2002.

[6] D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems (TOCS)*, 3(2):77–107, 1985.

[7] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 4(33):1–43, dec 2001.

[8] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the* 10^{th} *Symposium on High Performance Distributed Computing*, 2001.

[9] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, 1996.

[10] P. Felber, X. Défago, R. Guerraoui, and P. Oser. Failure detectors as first class objects. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99)*, pages 132–141, Edinburgh, Scotland, 1999.

[11] A. Gokhale and D. C. Schmidt. Measuring the performance of communication middleware on high-speed networks. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 306–317, New York USA, 1996. ACM SIGCOMM.

[12] R. Guerraoui and A. Schiper. Consensus service: A modular approach for building fault-tolerant agreement protocols in distributed systems. In *Proceedings of the* 26^{th} *International Symposium on Fault-Tolerant Computing (FTCS-26)*, pages 168–177, Sendai, Japan, 1996.

[13] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Computer Science Department, 1994.

[14] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. *Java Message Service*. Sun Microsystems, Inc., 901 San Antonio Road Palo Alto, CA 94303 USA, apr 2002.

[15] N. Hayashibara, A. Cherif, and T. Katayama. Failure detectors for large-scale distributed systems. In *Proceeding of the 1st Workshop on Self-Repairing and Self-Configurable Distributed Systems (RCDS), 21st IEEE Int'l Symp. on Reliable Distributed Systems (SRDS-21)*, pages 404–409, Osaka, Japan, 2002.

[16] M. A. Hiltunen and R. D. Schlichting. The cactus approach to building configurable middleware services. In *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, Nürnberg, Germany, 2000.

[17] T. A. Howes, M. C. Smith, and G. S. Good. *Understanding and Deploying LDAP Directory Services*. Macmillan Technical Publishing, 1999.

[18] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe: A group membership service for WANs. *ACM Transactions on Computer Systems*, aug 2002.

[19] P. Laumay, E. Bruneton, N. de Palma, and S. Krakowiak. Preserving causality in a scalable message-oriented middleware. *Lecture Notes in Computer Science*, 2218(311–??), 2001.

[20] C. P. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A toolkit for building fault-tolerant distributed applications in large scale. In *Workshop on Parallel and Distributed Platforms in Industrial Products*, San Antonio, Texas, USA, 1995. IEEE. Workshop held during the 7^{th} Symposium on Parallel and Distributed Processing, (SPDP-7).

[21] H. Miranda, A. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the* 21^{st} *International Conference on Distributed Computing Systems (ICDCS-01)*, pages 707–710, Phoenix, Arizona, USA, 2001. IEEE Computer Society.

[22] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: a fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, 1996.

[23] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. The design and implementation of Arjuna. Technical Report TR94-65, ESPRIT Basic Research Project BROADCAST, 1994.

[24] J. Postel. User datagram protocol. RFC 768, Internet Society (IETF), 1980.

[25] J. Postel. Transmission control protocol. RFC 793, Internet Society (IETF), 1981.

[26] H. Praveen, S. Arvindam, and S. Pokarna. Thunderbolt: A consensus-based infrastructure for loosely coupled cluster computing. In *Proceedings of the* 6^{th} *International Conference on High Performance Computing (HoPC)*, Calcutta, India, dec 1999.

[27] F. Reichenbach. Service SNMP de détection de faute pour des systèmes répartis. Diploma thesis, École Polytechnique Fédérale de Lausanne, Switzerland, feb 2002.

[28] M. Rose. The blocks extensible exchange protocol core. RFC 3080, Internet Society (IETF), 2001.

[29] M. Rose, G. Klyne, and D. Crocker. The application exchange core. RFC 3340, The Internet Society (IETF), 2002.

[30] P. Stelling, I. Foster, C. Kesselman, C.Lee, and G. von Laszewski. A fault detection service for wide area distributed computations. In *Proceedings of the* 7^{th} *Symposium on High Performance Distributed Computing*, pages 268–278, Chicago IL USA, 1998. IEEE.

[31] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[32] W. Yeong, T. Howes, and S. Kille. Lightweight directory access protocol. RFC 1777, Internet Engineering Task Force (IETF), mar 1995.