A Step Towards a New Generation of Group Communication Systems*

Sergio Mena, André Schiper, Paweł Wojciechowski

Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
1015 Lausanne, Switzerland
{first.last}@epfl.ch

Abstract. In this paper, we propose a new architecture for group communication middleware. Current group communication systems share some common features, despite the big differences that exist among them. We first point out these common features by describing the most representative group communication architectures implemented over the last 15 years. Then we show the features of our new architecture, which provide several advantages over the existing architectures: (1) it is less complex, (2) it defines a set of group communication abstractions that is more consistent than the abstractions usually provided, and (3) it can be made more responsive in case of failures.

1 Introduction

1.1 Context

Group communication has been widely argued to be an important enabling technology for building fault-tolerant applications in distributed systems [6]. Traditionally, applications can be made tolerant to crashes by replicating critical processes. In such a context, the group communication system (or middleware) manages the interaction between the process replicas across the network. The implementation of group communication middleware in a system with process crashes and unpredictable communication delay is a difficult task. Therefore, it is important to have a clean design of the group communication's architecture, with a well-understood set of programming abstractions provided by each component of the system.

In the dynamic group communication model, processes are organised into groups. The membership of a group can change over time, as processes *join* or *leave* the group, or as crashed processes are *removed* from the group. The current set of processes that are members of a group is called the *group view*.

* Research funded by the EPFL grant "Semantics-Guided Design and Implementation of Group Communication Middleware", by the Swiss National Science Foundation under grant number 21-67715.02, and partially by OFES under contract number 02.0328, as part of the IST MIDAS project (2001-37610).

Processes are added to and deleted from the group view via *view changes*, handled by a *membership* service. Communication to the members of a group is done by various *broadcast* primitives. The basic "reliable" broadcast primitive in the context of a view is called *view synchronous broadcast*, or simply *view synchrony*[1] [13]. The semantics of view synchronous broadcast can be enhanced by requiring messages to be delivered in the same order by all processes in the view. This primitive is called *atomic broadcast*[2] [13]. Moreover, different research groups distinguish between the *primary partition* membership and *partitionable* membership [13]. The discussion of these two models is outside the scope of this paper. In our work, we focus on the primary partition model, in which processes observe the same sequence of views. Primary partition membership is adequate for managing replicated servers, even in the case of link failures and/or network partitions.

The difficulty of implementing a group communication middleware can be formally explained by theoretical impossibility results, such as the impossibility of solving consensus in an asynchronous system when processes can crash [17]. These impossibility results can be overcome by strengthening a little bit the system model [10]. Even though many experimental group communication systems have been implemented during the last decade, so far the use of group communication middleware has not yet become a common practice when building fault-tolerant applications. We think that one of the reasons is the complexity of specifications of group communication services, which makes it difficult to understand the services provided by these systems. Another reason is the complexity of the systems itself.

1.2 Traditional Group Communication Architecture

In [13], Chockler *et al.* describe a comprehensive set of specifications of group communication services, which correspond to the most popular implementations. These specifications can serve as a unifying framework for the classification, analysis and comparison of the group communication systems that have been implemented over the last fifteen years.

The first observation we made is that all the implemented group communication systems we are aware of, adopt the same basic architecture, in which the group membership and view synchrony services are the basic components in the system. The guarantees provided by these two basic components are then used to implement other group communication services, e.g., atomic broadcast. We call this architecture the "traditional architecture".

---

[1] Basically, view synchrony ensures that between two consecutive views v and v', processes that are members of v and v' deliver the same set of messages broadcast to the group. View synchrony is sometimes called *virtual synchrony. View synchronous broadcast* is actually the best denomination, but we keep the term *view synchrony* to be consistent with the group communication literature.

[2] Atomic broadcast is also called *total order broadcast*.

1.3 Contribution: a New Architecture

In this paper we propose a new architecture with two key features that distinguish it from traditional architectures.

The first key feature is atomic broadcast (instead of group membership and view synchrony) as the basic component. The atomic broadcast component is then used to build other group communication services on top, e.g., group membership. Such an architecture has better separation of concerns. For example, the group membership service usually has to deliver new group views with guarantees that resemble those provided by the atomic broadcast. Therefore it seems logical for the atomic broadcast service to be more primitive than the group membership service. This architecture is formally supported by the new specification of group communication given in [32].

The second key feature of our new architecture is the absence of the view-synchrony service. This traditional service, which has a rather complex specification [13], is replaced by a new service called *generic broadcast* [29, 28]. Generic broadcast has a simpler specification than view-synchronous broadcast, but at the same time provides a more general service.

In our opinion, the reason for adopting the traditional architecture in the implementation of group communication systems seems more historical than justified by some strong arguments against other architectures. At the time when the first group communication systems (such as Isis) were built, it was not clear how to implement fault-tolerant atomic broadcast protocols without reconfiguration to exclude crashed processes. In later years, when the first papers appeared that suggested a different implementation of atomic broadcast (for example [9]), the traditional architecture had been already well established and the new implementations of group communication systems usually closely followed this initial approach.

In this paper, we compare different variants of the traditional architecture and argue that our new architecture is not only more elegant than the traditional architecture, but also has several advantages, which make it an interesting choice for designing and implementing new generations of group communication systems. The rest of the paper is organized as follows. Section 2 presents examples of the traditional group communication architecture. Section 3 describes our new architecture. Section 4 discusses the advantages of the new architecture compared to the traditional architecture. Finally, Section 5 concludes the paper.

2 Existing Group Communication Architectures

In this section we present the architecture of existing group communication systems. Since it is not possible (and also not really worth) to present the architecture of all group communication systems that have been implemented, we have selected here the most representative architectures. At the end of the section we abstract from the specific architectures and draw some general conclusions.

We have divided this section into two parts: *monolithic* and *modular* systems. As the name suggests, monolithic systems do not allow the system to be easily customized to the user needs; modular systems allow the user, using off-the-shelf components, to build the protocol stack that fits his/her needs.

Among all existing monolithic systems, we have chosen to present Isis [7, 6], Phoenix [25], RMP [34, 27], and Totem [2]. Among modular systems the most representative one is Ensemble [21]. There are many other group communication systems but their architecture overlap with the ones presented here. Transis [14], Relacs [3], and Newtop [16] architectures overlap with Totem and Ensemble. JavaGroups [4] is strongly inspired by Ensemble (it can even be configured to use an Ensemble stack). The group communication protocol suite implemented in the Appia framework [26] is also strongly inspired by Ensemble. The membership service presented in [23] uses a token based approach as in Totem or RMP.

2.1 Monolithic Systems

Isis. Isis was the first system to propose group communication [7, 8]. It is a monolithic *primary partition* system, i.e., when a network partition occurs, the computation can only proceed in one partition of the network, called the *primary* partition. The Isis architecture is depicted in Figure 1. The main layers are the following:[3]

- The *group membership* layer, which is responsible for maintaining the membership of groups. This layer handles *joins* (request to join the group) and *leaves* (request to leave the group). The layer also excludes processes that are suspected to have crashed. The group membership layer ensures that processes deliver the successive views in the *same total order*.
- Group membership does not provide any semantics for communication. For that reason, the group membership layer needs to be extended with a layer providing a semantics for the messages broadcast to the current group members. This semantics is called *view synchrony* (see Section 1).
- The upper layer provides *atomic broadcast*: it ensures that messages are delivered in the same order by all processes. Atomic broadcast is implemented using the view synchrony layer [8].

Phoenix. The Phoenix architecture [25] is a variation of the Isis architecture (Fig 2). The basic layer solves the *consensus* problem [10].[4] Membership (primary partition) and view synchrony are provided by the same layer: both the membership problem and view synchrony are solved using the underlying consensus layer. Similarly to the Isis architecture, atomic broadcast is provided on top of the view synchrony/membership layer.

The main limitation of Isis is to provide the membership service at the level of *processors*. In case of partition, this leads the service to kill all processes on

[3] The architecture corresponds to the protocol described in [8]. Since we do not discuss *causal order* in the paper, the Isis causal order protocol does not appear here.

[4] In the consensus problem, each process p_i starts with an initial value v_i, and all correct processes must agree on a common value v that is one of the initial values v_i.

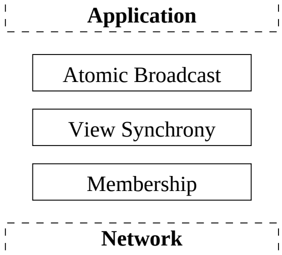

Fig. 1. Isis architecture

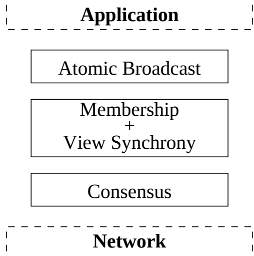

Fig. 2. Phoenix architecture

processors that are not in the primary partition. This drawback is prevented in Phoenix, which provides the membership service at the level of *processes*. This allows the computation to proceed in all partitions. Consider for example link failures leading to the following situation: the primary partition of some replicated service S is in some network component Π_1, and the primary partition of some other replicated service S' is in some other network component Π_2. A client process in Π_1 can read/update the service S and read S', while a client in Π_2 can read/update S' and read S.

RMP. RMP [34, 27] is another monolithic group communication system, whose architecture differs from the Isis and Phoenix architectures (see Figure 3). The RMP protocol has been influenced by Chang-Maxemchuk's atomic broadcast algorithm [11]. In RMP, the membership layer is split into two parts: *fault-free* membership and *fault-tolerant* membership.

The fault-free membership handles joins and leaves in the absence of failures, using the underlying atomic broadcast layer: joins/leaves are implemented using atomic broadcast. This totally orders joins/leaves with respect to any other application message that is issued using atomic broadcast, i.e., it ensures the *view synchrony* property in the absence of failures. However, the atomic broadcast protocol blocks in case of a process crash. The role of the fault-tolerant membership layer is to avoid blocking by excluding processes that are suspected to have crashed. The fault-tolerant membership protocol, based on a two-phase commit protocol [5] among the surviving processes, is completely different from the fault-free protocol. This fault-tolerant protocol has also the responsibility to ensure the view synchrony property, i.e., it orders view changes with respect to application messages that are atomically broadcast.

Totem. Unlike the architectures presented so far, Totem [2] – while being a monolithic architecture – is a representative of the systems based on the *partitionable membership* model.

Similarly to RMP, Totem uses an atomic broadcast algorithm based on a rotating token. Total order is provided by the middle layer of the architecture depicted in Figure 4 (the layer handles also flow control). The lower layer membership protocol, apart from detecting failures and defining views, recovers token and messages that had not been received by some members when failures occur. The top *recovery* layer completes the membership layer, by ensuring the (extended) view synchrony property.[5] When the membership layer is invoked, e.g., to exclude a process, it does not enforce the (extended) view synchrony property. This is ensured by the recovery layer.

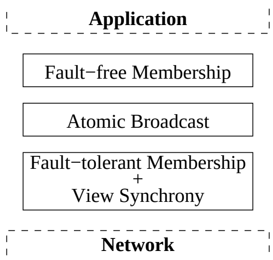

Fig. 3. RMP architecture

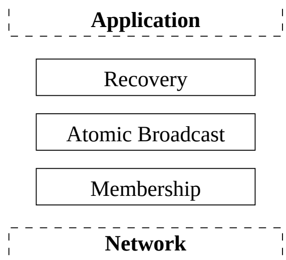

Fig. 4. Totem architecture

2.2 Modular Protocol Stacks

Unlike monolithic systems, modular systems allow users to customize the protocol stack to their specific needs. Horus [31] (the successor of Isis) and the re-implementation of Horus in the OCaml language called Ensemble [21] are the best representatives of modular group communication stacks. The idea is to use a set of off-the-shelf components and to compose them using the Horus/Ensemble framework to obtain a protocol stack with the functionalities customized to the user requirements. Similarly to Horus, Ensemble is based on the partitionable membership model. A sample Ensemble protocol stack is depicted in Figure 5. A few explanations are needed:

– A component, e.g., *stable*, can be placed at many places in the stack. The choice of the place has an impact on efficiency. For example, the role of the *stable* component is to detect messages stability.[6] When stability is detected by the *stable* component, an event is delivered to the layer below, and travels down from layer to layer until it reaches the bottom of the stack. At this point the event is bounced back, and travels up through the stack from component to component, until it reaches the top of the stack. The notification of stability occurs during the *upwards* travel of the event.

[5] Extended view synchrony [13] extends the view synchrony property, defined in the context of the primary partition model, to the partitionable membership model.

[6] A message is stable at a process when the process knows that the message has been delivered at all destinations.

– The application is not the uppermost layer in the stack. The reason is that it would take more time to convey events from the network level to the application. The most efficient layering leads placing components active in *normal* scenarios *below* the application, and components that handle *abnormal* scenarios *above*.

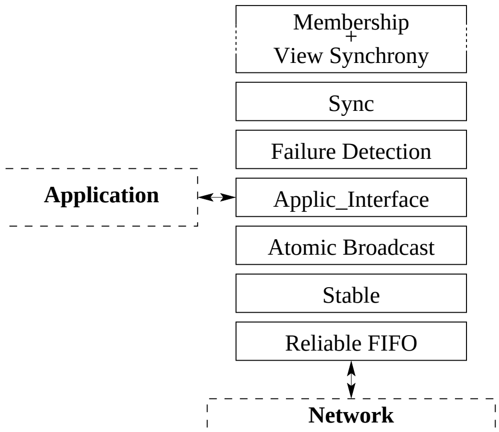

Fig. 5. Ensemble sample protocol stack

Apart from these generalities, here are comments about the Ensemble stack example depicted in Figure 5:

– The *atomic broadcast* component only orders messages in the absence of failures, or more precisely, when the system is stable (since Ensemble provides a partitionable membership service). Without additional membership layers, the different atomic broadcast protocols used would block in case of failures (e.g., upon crash, or disconnection).
– *Sync:* The layer implements a protocol for blocking a group during view changes, i.e., for preventing the broadcast of new messages during view changes.
– *Membership:* Actually, this is not a single layer, but a protocol suite, which includes various components, e.g., *merge*, *inter*, *intra*, etc. It is important to note that even though the membership component appears above the atomic broadcast component, it does not rely on it at all; a correct stack can have the membership components without the atomic broadcast component.

2.3 Discussion

The first observation from the above overview is that an architecture is necessarily influenced by the underlying algorithms. In other words, the architectures that have been presented differ because they rely on different algorithms (for membership, for view synchrony, for atomic broadcast, etc.). However, even though these architectures differ, they share some common features.

Group Membership and Failure Detection Are Strongly Coupled. *Failure detection* is a lower level mechanism than *group membership*. Failure detection gives notification of (possible) process failures (or disconnection) without worrying about inconsistencies (e.g., process p might suspect process r, whereas q might never suspect r). On the other hand, group membership gives *consistent* failure notification.[7]

However, none of the architectures that we have presented exploits this difference: group membership and failure detection are strongly coupled. In most of the architectures, the failure detection component does not even appear explicitly: it is completely hidden within the group membership component. Even in the architectures where the failure detection component is not hidden in the group membership, it directly interacts with the group membership, and only with it. In other words, other components learn about suspicions from the group membership component, not from the failure detection component. The group membership component acts as a failure detection component for the rest of the system.

Atomic Broadcast Algorithms Rely on Group Membership. A corollary of the previous observation is that, in all the above architectures, atomic broadcast algorithms rely on the group membership component; all these algorithms require the help of group membership to avoid blocking in the case of the failure of some critical process.

Basically these atomic broadcast algorithms operate in two modes: (1) a failure-free mode, and (2) a failure mode. A failure notification received from the group membership leads the protocol to switch from the failure-free mode to the failure mode. Here are two examples:

– In Isis and Phoenix, atomic broadcast is implemented using a fixed *sequencer* process. In the normal mode, the sequencer process attaches sequence numbers to messages that are atomically broadcast. However, the protocol blocks if the sequencer crashes. The notification of the failure of the sequencer is needed to prevent blocking, and to switch to the failure mode. In the failure mode the algorithm ensures that if one process has received a sequence number for some message m, then all correct processes receive the same sequence number for m. Once this is ensured, a new sequencer is chosen, and the algorithm returns to the normal mode.
– In RMP and Totem, processes form a logical ring and atomic broadcast is implemented using a rotating *token*. In the normal mode, the token is passed over the ring of processes. A process holding the token can attach a sequence

[7] The notion of *consistency* differs in the primary and in the partitionable membership service.

number to the messages it wants to broadcast. If one process crashes, the ring is broken, and the token may be lost. The failure mode is needed to recover from this situation.

This dependency of atomic broadcast on group membership is visible in the above stacks, where the membership component is *below* the atomic broadcast component. This is only partially true for RMP (see Figure 3), in which the dependency of atomic broadcast on group membership holds only in case of failures (failure-free membership is implemented using atomic broadcast). This dependency of atomic broadcast on group membership also holds in Ensemble, even though the atomic broadcast component is below the membership component in the stack in Figure 5: in Ensemble, as already explained, the layering of components does not reflect functional dependencies.

The Consensus Abstraction is Barely Used. When the consensus problem was defined in the early eighties [18], it was largely considered as a theoretical problem, with little practical relevance. Since then, the practical importance of consensus for solving problems such as atomic broadcast, (primary partition) group membership or view synchrony has been recognized. Nevertheless, except for Phoenix, no consensus component appears in the implementations.

Notice that this comment about consensus applies only to the primary partition systems, since the role of consensus in the context of partitionable group membership and extended view synchrony [13] (the counterpart of view synchrony in the context of partitionable membership) is not clear.

3 The New Architecture

We present now our new architecture. We proceed in three steps, starting with an overview at the same level of details as the architectures presented in Section 2 (allowing comparison). Then in Section 3.2, we present the augmented version of the architecture with a new key component: *generic broadcast*. Finally in Section 3.3, we describe the full version of the architecture with additional details.

3.1 Overview of the New Architecture

Figure 6 shows an overview of our new architecture. At this level of details, we can already see three important features:[8]

- Atomic broadcast does not rely on group membership, but *group membership relies on atomic broadcast.*
- There is no *view synchrony* component.
- Group membership and failure detection are decoupled.

[8] Note that Figure 6 does not mean that the application can only interact with the Group Membership component (the component just below the application).

Group Membership Relies on Atomic Broadcast and Not the Opposite. All the systems that have been described in Section 2 rely on atomic broadcast algorithms that require a perfect failure detector, i.e., a failure detector that makes no mistakes. This failure detector is denoted by $\mathcal{P}$ in [10]. The group membership service, when placed below atomic broadcast, emulates the perfect failure detector $\mathcal{P}$ by forcing incorrectly suspected processes to crash.

Instead, we propose to use an atomic broadcast algorithm requiring a $\diamond\mathcal{S}$ failure detector (much weaker than $\mathcal{P}$), which allows to make mistakes by suspecting correct processes: $\diamond\mathcal{S}$ allows even an unbounded number of wrong suspicions. Such an atomic broadcast algorithm is given in [10]: it is based on a sequence of instances of consensus (see the *consensus* component in Figure 6 below the total order broadcast component). This algorithm is able to work without blocking even if up to $f < n/2$ crashes occur. As a result, this algorithm does not have to rely on a group membership service.

Since the group membership component does not need to appear *below* the atomic broadcast component, it can be placed *above*: this means that group membership can be implemented using atomic broadcast, which is quite natural, since views need to be totally ordered. This generalizes the solution of RMP (Sect. 2.1). However, because of the limitations of the atomic broadcast algorithm used by RMP (it assumes a perfect failure detector, emulated by the membership service), RMP could use the solution only in the absence of failures: RMP's atomic broadcast relies on membership in case of failures.

It might appear to the reader that inverting the group membership component and the atomic broadcast component in the stack is just moving the complexity from one component to the other (the more complex component being the lowest in the stack). This is not true. It should be noted that any solution that implements (primary partition) group membership *below* atomic broadcast, actually has two algorithms to solve the same ordering problem: one specific solution to order membership changes, and one general (in the context of atomic broadcast) to order application messages. This only observation suggests that such architectures are not optimal.

There Is No View Synchrony Component. There is no view synchrony component in Figure 6. This component is replaced by a more powerful component, called *generic broadcast*, which is discussed below.

Group Membership and Failure Detection Are Decoupled. The strong coupling between failure detection and group membership in the architectures described in Section 2 was motivated by the atomic broadcast algorithms (requirement of a perfect failure detector emulated by the membership service). These architectures could not exploit the distinction between failure suspicion and membership exclusion (only process exclusions could be exploited by the atomic broadcast algorithm).

Decoupling group membership from failure detection has the following advantage: failure detections do not necessarily lead to process exclusion. This also means that decisions to exclude processes are no more taken by the group membership component. We come back to this issue below.

3.2 Augmented Version of the New Architecture

We introduce now the key component of our new architecture, namely *generic broadcast* (see Figure 7).

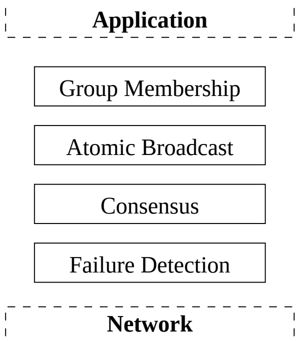

Fig. 6. New architecture: overview

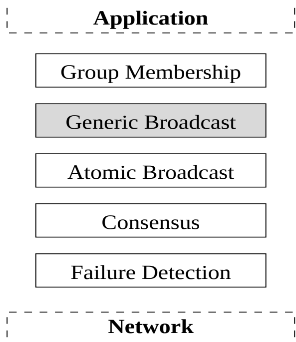

Fig. 7. New architecture with the Generic Broadcast component

Generic Broadcast Component. Generic broadcast is a powerful group communication primitive proposed recently [29, 30]. It is generic in the sense that the ordering of messages is defined by a conflict relation on the messages. If two *conflicting* messages m and m' are broadcast, then generic broadcast delivers them in the same order on all destination processes. However, if m and m' do *not* conflict, then generic broadcast does not order them (which is less expensive). So, if all messages conflict, then generic broadcast is equivalent to atomic broadcast. If no message conflicts, then generic broadcast reduces to reliable broadcast. As we explain below, generic broadcast favourably replaces view synchrony.

In terms of the implementation of our architecture, we assume here a thrifty implementation of generic broadcast that uses atomic broadcast [1]. In such a solution, atomic broadcast is not necessarily called in every run. Atomic broadcast is used only when conflicting messages are broadcast (see [1] for an extended discussion of the notion of *thrifty* implementation of generic broadcast).

Active and Passive Replication. Since a group communication middleware is supposed to provide abstractions for the replication of critical components, it is natural to confront the abstractions provided so far with the needs of replication techniques. Our preliminary architecture (Fig. 6) provides atomic broadcast, which allows us to implement active replication [33], also called state machine approach (in active replication, the client requests are sent to all servers using atomic broadcast, and every server processes the request).

Atomic broadcast is not needed in passive replication. Instead, view synchrony provides the right abstraction, see for example [20]. However, our new stack does not provide such an abstraction. We illustrate in the next section how generic broadcast can be used in place of view synchrony. More generally, as shown in [32], view synchrony does not need to be considered as a basic abstraction. View synchrony follows rather from adequate specifications of dynamic group communication [32].

Generic Broadcast instead of View Synchrony for Passive Replication. In passive replication, the client sends its request to only one server, the *primary*. Only the primary processes the client request; before sending the response back to the client, the primary updates the state of the backups. This is done by an *update* message, sent from the primary to the backups. The standard solution consists in relying here on *view synchrony*.

With generic broadcast,[9] the solution consists in considering two types of messages (Fig. 8): (1) *update* messages, and (2) *primary change* messages. The *update* messages are used by the primary to update the state of the backups. The *primary change* messages are used by the backups to change the new primary, when the current primary is suspected to have crashed. A *primary change* message does not lead to the exclusion of the old primary, which remains in the view. If the primary has actually crashed, a new view will be installed to exclude it after a very long timeout (see *Monitoring Component*, Section 3.3).

The conflict relation between *update* and *primary change* messages is as follows:

	update	primary change
update	*no conflict*	*conflict*
primary change	*conflict*	*conflict*

This conflict relation ensures that (1) *primary change* messages are totally ordered, (2) *update* messages are totally ordered with respect to *primary change*, and (3) *update* messages are not ordered with respect to other *update* messages. For illustration, consider a replicated server with three replicas s_1, s_2, s_3 (which define the group s) and the following scenario (Figure 8):

- The server s_1 is initially the primary.
- At time t, s_1 receives a client request, processes it, and generic-broadcasts the *update* message to the group s.

[9] In this example we have to assume FIFO generic broadcast, i.e., the FIFO point-to-point property in addition to the ordering properties of generic broadcast. The same FIFO property is required in the context of the solution based on view synchrony.

– Approximately at the the same time t, server s_2 suspects s_1 to have crashed, and generic-broadcasts the *"primary-change(s_1)"* message to the group s. Upon delivery of this message, all servers (including s_2) modify their view from $[s_1; s_2; s_3]$ to $[s_2; s_3; s_1]$, which leads the servers to consider s_2 to be the new primary.[10]

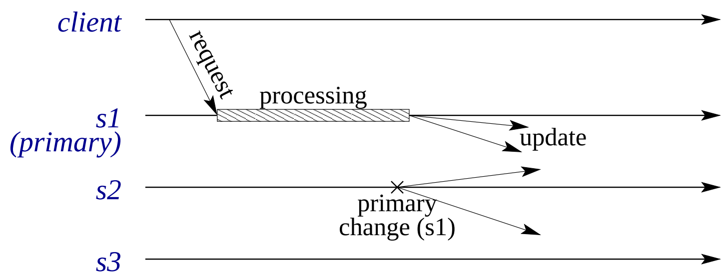

Fig. 8. Generic broadcast for passive replication

Since these two messages conflict, we have only two possible outcomes:

1. All members of s deliver the *update* message before the *primary-change* message.
2. All members of s deliver the *primary-change* message before the *update* message.

In case 1, the primary change occurs logically *after* the handling of request *req* by s_1. In case 2, the primary change occurs logically *before* the handling of the request. This means that the processing of the request by s_1 must be ignored. The client will timeout, learn that s_2 is the new primary, and reissue its request to s_2.

3.3 Full Version of the New Architecture

The full version of our architecture, which includes all components and all interfaces between components, is given in Figure 9. The additional components are:

– the *reliable channel* component,
– the *monitoring* component.

[10] Views are here *lists* of processes, rather than *sets* of processes. The primary is the process at the head of the list. Note that the delivery of the *primary-change(s_1)* message does not lead to the exclusion of s_1.

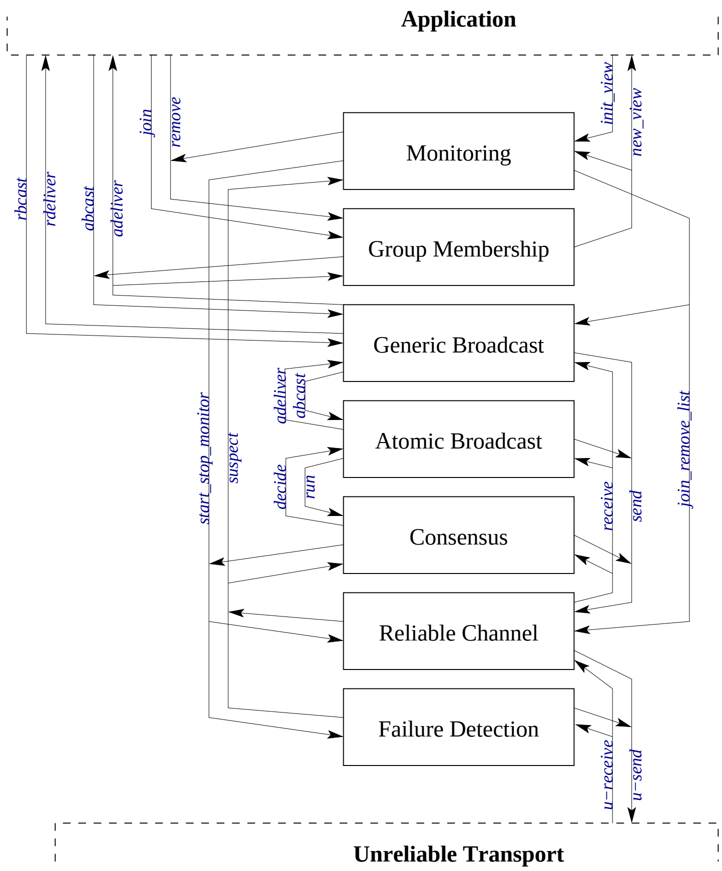

Fig. 9. New architecture: full version

Note that in Figure 9, the operations on the generic broadcast component are called *abcast* (invocation of atomic broadcast) and *rbcast* (invocation of reliable broadcast).[11] The conflict relation is the following:

	rbcast	abcast
rbcast	*no conflict*	*conflict*
abcast	*conflict*	*conflict*

In other words, in the context of the passive replication example, *rbcast* should be used for the *"udpate"* message, and *abcast* for the *"new primary"* message. Of course, generic broadcast can be initialized with a different conflict relation table.

We explain now briefly the role of the *reliable channel* and *monitoring* components.

Reliable Channel Component. The *reliable channel* component ensures the following property: if a correct process p sends message m to some correct process q, then q eventually receives m. This abstraction can be easily implemented on top of TCP [15].

Monitoring Component. In our architecture, the decision to exclude a suspected process from the membership is not made by the group membership component.[12] The decision is made by the monitoring component, which then calls the *remove* operation of the *membership* component.

The separation of concerns between the *failure detection* component and the *monitoring* component allows for very flexible policies. On the one hand, the *consensus* component of process p could ask the *failure detection* component to use a *small* timeout value (e.g., in the order of seconds) to suspect some other process q. Typically, this suspicion would *not* lead to the exclusion of q. On the other hand, the *monitoring* component of p might ask the *failure detection* component to use a large timeout value (e.g., in the order of minutes) to suspect q. Here a suspicion would lead the *monitoring* component to call the *membership* component to *remove* q. However, to make such a decision, the monitoring component may also interact with the *monitoring* component of other processes, and for example decide on the removal of q only after having learned that a threshold of other processes also suspect q.

Still another exclusion policy can be expressed, which is relevant when process p sends message m to process q. The *reliable channel* component at p buffers m, until $ack(m)$ is received from q (which acknowledges reception of m by q). If q crashes, m might stay in p's buffer forever. In this case, the only way to discard

[11] See [32] for a precise specification.

[12] The operations on the membership component are *join* – to add a process to the group, and *remove* – to remove a process from the group (including itself).

m is to exclude q from the membership (if q is excluded from the membership, there is no more obligation for q to deliver m, i.e., m can be safely discarded). This is called *output-triggered* suspicion in [12]. The *monitoring* component can exclude processes based on output-triggered suspicions (which should be based on long timeout values).

4 Assessment of the New Architecture

We stress now on the advantages of the new architecture compared to the traditional architectures presented in Section 2.

4.1 Less Complex Stack

With traditional architectures, the ordering problem is solved in two places: (1) within the group membership component for views, and (2) within the atomic broadcast component for messages.[13] From a conceptual point of view this is not optimal, and introduces an unnecessary complexity. This redundancy has disappeared in the new architecture, where the ordering problem is solved only once (in the atomic broadcast component).

Actually, in the traditional architectures the ordering problem is even solved in a third place, namely in the view synchrony component, which orders messages with respect to view changes. In our new architecture, this additional ordering problem is also solved in the same place, namely in the atomic broadcast component. Indeed, when the generic broadcast component detects a message conflict (e.g., between a reliable broadcast and atomic broadcast view change message), then it calls the atomic broadcast component. The details can be found in the thrifty generic broadcast algorithm [1].

Altogether, from the point of view of the ordering problem, the new architecture is less complex than traditional architectures. Smaller complexity usually leads to easier maintenance.

4.2 More Powerful Stack (Provides More Functionalities)

The new suite of components provides functionalities which are not present in traditional stacks. The prominent example is generic broadcast, which extends the ordering provided by view synchrony. Consider for example a replicated service managing client bank accounts, with deposit and withdrawal operations (withdrawal does not allow to withdraw more than available). Both classes of operations update the state of the server, but deposit operations are commutative, i.e., they do not need to be ordered with respect to themselves. This ordering typically can be solved using generic broadcast. Traditional stacks do not provide any specific solution: atomic broadcast would have to be used both for deposit and withdrawal operations. This would induce a non-necessary overhead.

[13] In RMP, ordering is performed in two different places only in case of failures.

On a more minor issue, the fact that failure suspicions can be generated in two distinct places is not without benefit. Depending on the context, the monitoring component can take the decision to exclude a process from the membership either (1) based on notification from the failure detector component, or (2) based on notifications from the reliable channel components, or (3) it could wait for notifications from both components.

4.3 Higher Responsiveness

Group communication allows the implementation of fault-tolerant replicated services. Performance of group communication is usually measured in failure-free executions. However, performance of group communication in case of failures often is equally important.

Consider for example the latency of atomic broadcast, i.e., the time elapsed between the atomic broadcast of m and the first delivery of m. In case of failures, the timeout used to detect failures represents an important part of this latency. So, reducing the latency in case of failures requires failure detection timeouts to be as small as possible. However, reducing failure detection timeouts increases the probability of false suspicions. Decoupling failure suspicions from process exclusions plays here an important role.

In traditional architectures, wrong failure suspicions have a high cost: the cost of excluding the wrongly suspected processes, followed by the cost of the join operation (with the costly state transfer operation) in order to include again the process in the membership. This has forced traditional systems to adopt large failure detection timeout values. In our stack, where failure suspicions are decoupled from exclusions (i.e., false suspicions lead to a small overhead), timeouts can be chosen to be smaller. This leads to a gain in efficiency in case of failures, e.g., to higher responsiveness.

4.4 Minor Efficiency Issue

Traditional systems have another responsiveness problem, namely in the context of view changes. This problem is not related to failures, since view changes may be triggered by *join* requests, and *remove* requests that are not exclusions. The traditional solution in the context of membership changes ensures that messages broadcast before the membership change are delivered before the membership change takes place. This property is called *sending view delivery* (see [13]). However, in order to ensure this property without discarding messages, processes must stop sending messages while the membership change protocol is running (see for example the *Sync* layer of Ensemble, Section 2.2). To prevent this undesirable *blocking* problem, which reduces responsiveness, alternate and more complex solutions to handle membership changes have been proposed [19, 24]. These solutions implement a weaker property called *same view delivery* [13]. The implementation based on generic broadcast does not lead to blocking: the solution “naturally” implements the *same view delivery* property without additional complexity [32].

5 Conclusion

Existing group communication systems (GCS) can be classified according to two dimensions: (1) the *membership model* dimension, and (2) the *structuring* dimension. The *membership model* dimension allows the classification of GCS as either (i) *primary partition* GCS, or (ii) *partionable membership* GCS. The *structuring* dimension allows the classification of GCS as either (i) *monolithic* or (ii) *modular*. Isis, falls into the category *primary partition/monolithic*, while Ensemble falls into the category *partitionable/modular*.

This paper has introduced a third dimension: the *protocol* dimension. With respect to this third dimension, existing GCS can be characterized as *GM-VS*:[14] (1) membership is the basic component in the stack, and (2) view synchrony is the basic communication abstraction. The paper has presented an alternate solution that could be called *AB-GB*[15] based: (1) atomic broadcast is the basic component, (2) no view synchrony as such is provided, and (3) the GCS provides generic broadcast (instead of view synchrony) as a more powerful abstraction.

We have started the implementation of this new architecture, using two different protocol composition frameworks: Appia [26] and Cactus [35, 22]. The two implementations share the same protocol code at each module, and differ only in the way interactions (events) are routed across modules in each of the frameworks.

References

1. M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC'2000)*, October 2000.
2. Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P.Ciarfella. The Totem Single-Ring Ordering and Membership Protocol. *ACM Trans. on Computer Systems*, 13(4):311–342, November 1995.
3. O. Babaoglu, R. Davoli, L. Giachini, and M. Baker. Relacs: A communication infrastructure for constructing reliable applications in large-scale distributed systems. In *Proceedings of the 28th Hawaii Interntional Conference on System Sciences*, volume II, pages 612–621, Jan 1995.
4. Bela Ban. *JavaGroups 2.0 User's Guide*, Nov 2002.
5. P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Distributed Database Systems.* Addison-Wesley, 1987.
6. K. Birman. The Process Group Approach to Reliable Distributed Computing. *Comm. ACM*, 36(12):37–53, December 1993.
7. K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. on Computer Systems*, 5(1):47–76, February 1987.
8. K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. on Computer Systems*, 9(3):272–314, August 1991.

[14] *G*roup *M*embership - *V*iew *S*ynchrony.
[15] *A*tomic *B*roadcast - *G*eneric *B*roadcast.

9. T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. In *proc. 10th annual ACM Symposium on Principles of Distributed Computing*, pages 325–340, 1991.
10. T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, 1996.
11. J. M. Chang and N. Maxemchuck. Reliable Broadcast Protocols. *ACM Trans. on Computer Systems*, 2(3):251–273, August 1984.
12. B. Charron-Bost, X. Défago, and A. Schiper. Broadcasting messages in fault-tolerant distributed systems: the benefit of handling input-triggered and output-triggered suspicions differently. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 244–249, Osaka, Japan, October 2002.
13. Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):1–43, December 2001.
14. Danny Dolev and Dalia Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, 1996.
15. Richard Ekwall, Péter Urbán, and André Schiper. Robust TCP connections for fault tolerant computing. In *Proc. 9th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 501–508, Chung-li, Taiwan, December 2002.
16. Paul D. Ezhilchelvan, Raimundo A. Macedo, and Santosh K. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *International Conference on Distributed Computing Systems*, pages 296–306, 1995.
17. M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of ACM*, 32:374–382, April 1985.
18. M.J. Fischer. The consensus problem in unreliable distributed systems (A brief survey. In *Proc. Int. Conf. on Foundations of Computations Theory*, pages 127–140, 1983.
19. R. Friedman and R. van Renesse. Strong and Weak Virual Synchrony in Horus. In *15th IEEE Symp. on Reliable Distributed Systems (SRDS-15)*, pages 140–149, Niagara-on-the-Lake, Ontario, Canada, September 1996.
20. R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68–74, April 1997.
21. Mark Hayden. The Ensemble system. Technical Report TR98-1662, Department of Computer Science, Cornell University, January 8, 1998.
22. Jun He, Matti A. Hiltunen, Mohan Rajagopalan, and Richard D. Schlichting. Providing transparent qos customization for CORBA objects, 1997.
23. Matti A. Hiltunen and Richard D. Schlichting. A configurable membership service. *IEEE Transactions on Computers*, 47(5):573–586, 1998.
24. J. Sussman I. Keidar and K. Marzullo. Optimistic virtual synchrony. In *19th IEEE Symp. on Reliable Distributed Systems (SRDS-19)*, pages 42–51, Nurnberg, Germany, October 2000.
25. C. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks.* PhD thesis, Federal Institute of Technology, Lausanne (EPFL), 1996.
26. Hugo Miranda, Alexandre Pinto, and Luís Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of The 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 707–710, Phoenix, Arizona, USA, April16–19 2001. IEEE Computer Society.
27. Todd Montgomery. Design, implementation, and verification of the reliable multicast protocol. Master's thesis, West Virginia University, Dec 1994.

28. F. Pedone and A. Schiper. Handling Message Semantics with Generic Broadcast Protocols. *Distributed Computing*. Submitted for publication.
29. F. Pedone and A. Schiper. Generic Broadcast. In *13th. Intl. Symposium on Distributed Computing (DISC'99)*. Springer Verlag, LNCS 1693, September 1999. Extended version to appear in ACM Distributed Computing, 2002.
30. F. Pedone and A. Schiper. Handling Message Semanticas with Generic Broadcast Protocols. *Distributed Computing*, 15(2):97–107, april 2002.
31. Robbert Van Renesse, Kenneth P. Birman, Bradford B. Glade, Katie Guo, Mark Hayden, Takako Hickey, Dalia Malki, Alex Vaysburd, and Werner Vogels. Horus: A flexible group communications system. Technical Report TR95-1500, Department of Computer Science, Cornell University, Apr 1996.
32. André Schiper. Dynamic Group Communication. Technical Report, EPFL, March 2003.
33. F.B. Schneider. Replication Management using the State-Machine Approach. In Sape Mullender, editor, *Distributed Systems*, pages 169–197. ACM Press, 1993.
34. Brian Whetten, Todd Montgomery, and Simon M. Kaplan. A high performance totally ordered multicast protocol. In *Dagstuhl Seminar on Distributed Systems*, pages 33–57, 1994.
35. Gary T. Wong, Matti A. Hiltunen, and Richard D. Schlichting. A configurable and extensible transport protocol. In *INFOCOM'01*, April 2001.