Advances in the Design and Implementation of Group Communication Middleware

Daniel Bünzli, Rachele Fuzzati, Sergio Mena, Uwe Nestmann*
Olivier Rütti, André Schiper, and Paweł T. Wojciechowski**

Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
<firstname>.<lastname>@epfl.ch

Abstract. Group communication is a programming abstraction that allows a distributed group of processes to provide a reliable service in spite of the possibility of failures within the group. The goal of the project was to improve the state of the art of group communication in several directions: protocol frameworks, group communication stacks, specification, verification and robustness. The paper discusses the results obtained.

1 Introduction

Group communication is a programming abstraction that allows a distributed group of processes to provide a reliable service in spite of possible failures within the group. Group communication encompasses broadcast protocols (e.g., reliable broadcast, atomic broadcast), membership protocols, and agreement protocols.Group communication is a middleware technology, lying between an application layer and a transport layer. Developing and maintaining a group communication middleware is a non-trivial, error-prone and complex task. In this context, the goal of the project was to improve the state of the art in several directions: flexibility, reusability, formalization, verification and validation. Due to space constraints, a detailed presentation of all the contributions was not possible. The paper gives an overview of the results obtained, structured as shown in Figure 1. Details can be found in the referenced papers.

Protocol frameworks: Flexibility and reusability of a middleware layer—and of any piece of software—can be achieved by decomposing the middleware into protocols that can be assembled. The glue that allows to assemble the protocols is called a protocol *framework*. The features of a protocol framework are essential to the protocol composer: they can make life more or less easy. Typically, adequate features can reduce the errors at assembly time, i.e., when the protocols are glued together to build a group communication middleware. Protocol frameworks are addressed in Section 2, where we start by presenting the features of traditional protocol frameworks, before presenting the novel aspects that have been designed and implemented within the project.

* Now at Berlin University of Technology, 10587 Berlin, Germany.
** Now at Poznań University of Technology, 60-965 Poznań, Poland.

J. Kohlas, B. Meyer, and A. Schiper (Eds.): Dependable Systems, LNCS 4028, pp. 172–194, 2006.

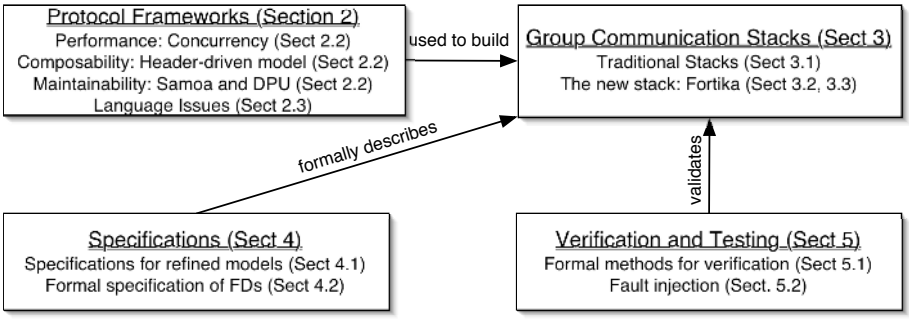

Fig. 1. Structure of the paper

Group communication stacks: A good protocol framework is essential to achieve flexibility and reusability, but is by itself not sufficient. It is also required to identify the "right" components (or protocols). In the context of group communication this is a difficult task, because of the difficulty of the problems to solve. Components derive from algorithms that solve these problems. In Section 3 we explain that the traditional architecture of a group communication middleware has considered *group membership* as one of its most basic components. We discuss the deficiencies of this choice and propose a different architecture. We believe that the new architecture has a much better chance to be flexible, i.e., to adapt to a changing environment. As a consequence, the components are much more likely to be reusable.

Specifications: In group communication, formalization plays an important role at two levels: *(i)* at the level of the specification of the problems to be solved, and *(ii)* at the level of solving these problems. At the specification level, we need a precise characterization of the desired properties of group communication primitives or services. At the solution level, we need to characterize the assumptions that allow us to solve the problems. These issues have been addressed in the past. However, it is only for static groups and in the so-called crash-stop model (in which processes do not recover after a crash) that specifications of group communication are widely accepted and agreed upon. Specifications for dynamic group communication have been proposed, but they differ significantly from static specifications, which is not satisfactory. In Section 4 we show that it is possible to specify dynamic group communication in such a way that, if the group happens to be static, then we obtain the well-known static specifications. We also address specifications of group communication in the crash-recovery model (in which processes have access to stable storage to save their state). Specifications for the crash-recovery model have been proposed in the past, but they fail to capture the fundamental difference between the crash-stop and the crash-recovery model. In Section 4 we discuss how this issue can be addressed. Section 4 also addresses an important formalization issue related to solving group communication problems. The concept of *unreliable failure detectors* has been introduced some time

ago. The section proposes a fresh look at failure detectors by representing them via transition systems (as known from operational semantics), with the goal of bringing the definitions closer to our formal reasoning techniques.

Verification and testing: Section 5 is devoted to verification and testing. Consensus is one of the most fundamental problems in group communication. Many of the algorithms for solving consensus have been described in pseudo code, and their proofs use concepts that are sometimes not rigorously defined. Here, we propose a formal approach to the proof of a rigorously defined consensus algorithm. Finally, Section 5 describes an experiment conducted on our new *Fortika* group communication middleware that was build within the project. Fault injection has been used to study how *Fortika* reacts to memory and network corruption—faults that *Fortika* was not designed to handle. The experiments allowed us to identify a few weaknesses and to correct them. The result is a very robust group communication middleware.

2 Protocol Frameworks

We first present the most significant existing frameworks. Then, we discuss some novel ideas that advance the state of the art of protocol composition, covering programming models, concurrency support and dynamic protocol update. Finally, we study languages to ensure the safe usage of concurrency support.

2.1 Existing Frameworks

Protocol frameworks are programming tools to build complex middleware out of simpler off-the-shelf building blocks, called *protocols*. This modular approach yields advantages such as customization, code reuse, extensibility, and ease of maintenance. Altogether, this eases the implementation of group communication middlewares (and, in turn, the implementation of fault tolerant applications).

Most of the existing frameworks are based on an event-driven programming model. In such a model, a protocol is structured as a set of events, handlers, bindings, and a state that can be private or shared with other protocols. Events carry data and are triggered by protocols. The handlers contain the code of the protocol and can modify the state and trigger events. A binding is a mapping between an event type and one or several event handlers. When a handler triggers an event, all handlers bound to its type are executed thereby possibly triggering new events. Composing protocols consists in binding events of a protocol to handlers of other protocols [14].

Cactus. Cactus [2] is an evolution of the x-kernel [4] protocol framework. Composition with x-kernel was strictly layered, i.e., a protocol could only communicate with the protocols immediately above or below. Cactus introduces a finer-grain level of composition where several *microprotocols* can be assembled to form an x-kernel protocol. Each microprotocol can communicate with any other microprotocol in the same protocol using events.

Cactus supports concurrent execution of its microprotocols. Concurrency inside composite protocols requires synchronization policies in order to preserve the properties provided by the composite protocols. The C version of Cactus offers basic guarantees in the presence of concurrency, such as atomic execution of handlers. In the Java version of Cactus, however, it wholly depends on the programmer, who must implement the required synchronization policy using standard language facilities (such as locks, semaphores, monitors, etc...). This approach, besides being tricky and error-prone, harms modularity, since a composer has to adapt the code of the microprotocols composed in order to come up with an adequate concurrency management.

Appia. Appia [1] is a re-engineering of the Ensemble group communication toolkit [3]. In Ensemble, programmers can compose protocols in layered stacks where there can only be one protocol per level in the composition. In contrast, Appia allows more flexible composition, where there may be several protocols per level.

Appia features a validation check for composition. Every protocol declares which event types it accepts, requires and provides. At composition time, Appia verifies that all required events are provided by some protocol, rejecting all compositions that do not pass this check. However, the fact that the direction of events is not taken into account makes this verification superficial.

Unlike Cactus, Appia does not allow concurrent execution of protocols. All events are dispatched by a single-threaded scheduler. This frees the protocol programmer from the burden of dealing with concurrency, deadlocks, etc. However, this absence of concurrency prevents Appia from making the most of high-performant systems (e.g., multi-processor platforms).

2.2 Novel Aspects

Our project brought up new programming abstractions that either improve over, or provide an alternative to existing protocol frameworks. First we describe a new *header-driven* programming model to program protocols in frameworks. This model solves well-known composition problems of the event-driven model and thereby enables the use of powerful composition languages (e.g. ML module systems) to structure protocols and stacks. Then, we describe a runtime system to support concurrency inside a protocol stack without the need to make the protocols themselves aware of concurrency. Finally, we describe SAMOA—a novel protocol framework that implements the runtime system, and provides other interesting features such as dynamic protocol update.

A Header-Driven Programming Model. All recent frameworks use a general-purpose event-driven programming model to manage interactions between protocols. However in complex compositions, where protocols offer their service to more than one protocol, the one-to-many interaction scheme of events introduces composition problems by mixing up the targets to which data should be delivered. In other words, protocols may receive events with data that is not

targeted at them. This problem compromises the use of powerful composition languages on top of an event-driven model, because *ad hoc* mechanisms need to be introduced to "route" events to the right protocols. Moreover, the event model doesn't properly handle *peer interactions*, where a protocol interacts with its peer running on another node by using the service of a lower-level protocol. The way events handle this ubiquitous pattern is critical in three ways: (1) it is complex, because invariants known at design time need to be enforced by the composer; (2) it is obscure, because the indirections introduced by the events hide, in the code, the logical structure of peer interactions; (3) it is unsafe, because misbindings can lead to runtime type errors or erratic behaviour. To solve these problems we propose in [6] a novel and simple alternative that shifts the driving force behind interactions from events to the headers (data) they carry.

In the event-driven model, protocols typically encapsulate communication data for their peers in the messages and headers that are carried by events. A message is a list of headers and a header is a typed container for data. Protocols often use a single event handler to manage the reception of messages. Nevertheless, some protocols need to get different kinds of data via this single handler. In order to do so they introduce a *tag name* in the header to indicate the kind of information being transmitted. Since a header usually remains internal to a protocol and its peers, it is not restrictive to impose that each header shall be *named* and that each name shall be *declared by at most one protocol* in a composition. A protocol composition satisfying these constraints has the following interesting property. If we look at the names of a message's sequence of headers, we can approximately see the sequence of protocols—the route—that will handle the message when it is processed by the composition of protocols. In other words the message's sequence of headers drives its processing in the composition. The event model prevents us from exploiting this property. Thus instead of having events at the core of our interaction scheme we should have *headers*. This is the essence of our proposal.

The essential ingredients of a *header-driven model* are headers and messages. As before, a message is a list of headers. But headers are *named* containers carrying statically typed data. To construct a header, its name must be defined. A header handler defines a header name and associates a computation to the deconstruction of every message that starts with this name. Message dispatch is the interaction scheme, it deconstructs messages. When a message is dispatched, the unique header handler corresponding to the head of the message is invoked with the head's data and the tail of the message as arguments. Compared to the event model we can say that (1) header handlers replace event handlers, (2) message dispatch replaces event triggering, and (3) the event binding mechanism is dropped.

The resulting *header-driven* model has several advantages. It solves the composition problems of the event model, it simplifies inter-protocol dependencies and hence the task of composing protocols, and it concisely handles peer interactions and explicitly reveals their logical structure (no binding indirections). Moreover, the *header-driven* model provides better static typing, which avoids

runtime type errors and erratic behaviours that can occur in the event model. Our approach was validated by a proof-of-concept implementation [11].

Automatic Concurrency Control for Protocol Stacks. Implementing atomic processing of concurrent messages in a protocol stack is notoriously hard and error-prone. *Atomic transactions* can greatly help programming. Every fresh message is processed by a new transaction with the guarantee of *isolation*—a property known from standard ACID (Atomicity, Concurrency, Isolation and Durability) transactions[1]. The usual implementation of atomic transactions depends however on *rollback-recovery*—if some operations of concurrent transactions conflict with respect to isolation, then the transactions are started again. Rolling back some of the *input/output (I/O)* effects is problematic, e.g. resending messages that have been output to the network may confuse the distributed protocol. To evade this problem, we have designed *rollback-free* concurrency control algorithms for protocol frameworks [28]. The algorithms implement runtime *versioning and scheduling* of transaction operations.

The basic idea of versioning is the following. Tickets (or *versions*) are assigned to isolation-only transactions (called *tasks*) that allow them to acquire *verlocks*, i.e., versioning locks protecting isolation-critical operations. On task creation, a task obtains incremented version values for all verlocks that it wants. The task can acquire a verlock only when the verlock's counter has reached the version count. The counters are monotonically increasing counters, one per verlock. The counter is increased when the verlock has been released by a given task for *the last time*.

In [28], we described three variants of versioning algorithms (*Basic*, *Bound*, and *Route*), which differ in the precision of detecting when verlocks are actually requested for the last time. To detect this moment, upon a task creation, the algorithms require some data about the task to be passed as the argument. Different variants of the algorithms can be characterized as follows:

- *Basic*: it gives an almost serial execution, however only verlock names must be known *a priori* (they can be inferred statically, as described in Section 2.3);
- *Bound*: it requires a least-upper-bound (supremum) on verlock access to be known *a priori*; in general, this variant allows for more parallelism than *Basic*, but it performs like *Basic* if supremum cannot be reached;
- *Route*: it allows for even more parallelism than *Bound*, however it demands *a priori* a complete tree of potential accesses to verlocks within scope of a task, where a branch in the tree corresponds to a thread of execution.

The SAMOA Protocol Framework and Dynamic Update. We have developed *SAMOA*, a novel protocol framework that improves over the existing protocol frameworks in two respects. Firstly, *type-safe* dynamic protocol (re)binding guarantees that no runtime errors can happen due to protocol interactions. Secondly, isolation-only, rollback-free transactions (or *tasks*) make it

[1] This property is similar to *atomicity* in the programming language community.

easier for programmers to encode concurrent protocols that may have irrevocable I/O effects; the implementation of tasks uses concurrency control algorithms described above. An experimental implementation of SAMOA as a package in Java is available [20]. Some of the key features of SAMOA are the possibility to load protocols *on-the-fly* and to dynamically bind and unbind protocols. Based on these features, we have implemented efficient mechanisms for *dynamic protocol update*. The problem of dynamic protocol update has been addressed in [27]. Essentially, we must guarantee global service availability and correctness while a distributed update operation takes place. To validate these ideas, we have implemented the *Adaptive Group Communication* (AGC) middleware in which protocols can be replaced dynamically. The AGC middleware uses the Fortika group communication protocols that supports both the crash-stop and crash-recovery models (see Sections 3 and 4.1).

2.3 Language Issues

The use of isolated tasks in our implementation of SAMOA in Java is not *safe*, i.e., the programmer must carefully determine and declare certain data about tasks for the concurrency control algorithms to work correctly. Below we describe the design of a safe programming language that removes this drawback.

Another problem with the existing protocol frameworks that support concurrency (such as SAMOA and Cactus) is that it is not possible to reuse protocols that contain any synchronization constructs, such as spawning a new task, without inspecting the code of these protocols (since these constructs may need to be removed in the new stack). To remove this drawback, we have proposed to *separate* the synchronization code and the protocol code. In the end of this subsection, we discuss the design of two languages for this separation. These languages could become an integral part of any future protocol frameworks.

Language Design For Isolated Tasks with I/O Effects. To spawn a new task, the SAMOA programmer uses a construct `isolated` R e, where e is a concurrent, isolated task. In the simplest case (Basic versioning algorithm), argument R is just a list of service names that *may* be requested by the task e. Unfortunately, a wrong argument R compromises safety (and therefore also correctness!). How could one make this construct *safe* for programmers? The idea is to design a *type system* that can verify at compile time whether the argument R is correct. A type system—implemented as part of the compilation tool—can prove that *any* program execution preserves given properties expressed as types (informally: *"well typed programs can't go wrong"*).

In [26,25], we designed a *typed language* for expressing rollback-free transactions (tasks) in modular protocols. It has a construct `isolated` R e to spawn a new task e, where task expression e is executed by a new thread. Any return result of this evaluation is ignored—the only visible outcome of task execution are any *data and I/O effects*. Any threads spawn by e are part of the *same* task. The concurrent execution of tasks satisfies the *isolation property*. The argument R in `isolated` R e is a list of verlock names. A new verlock x can be created

with `newlock` x `:` m `in` e, where m is an *effect type* of a single verlock. We normally create a fresh verlock for each communication channel (I/O effects), and each data structure (memory change effects). Verlocks can be shared by concurrent tasks.

We can use verlocks in the expression `sync` e' e. The expression has a semantics that is roughly similar to `synchronized` in Java, i.e., expression e' is evaluated first and yields verlock x (of type m), which is then acquired when possible. Expression e is then evaluated to some v. Finally, the verlock is released and v returned by the whole expression. There is, however, an important difference. What is a locking strategy? Or, when exactly the verlock x is acquired when executing the expression `sync` x e? Verlock x of type m is *acquired* when two conditions are satisfied: (1) the effect m caused by e does not conflict (with respect to *isolation*) with the effects of other concurrent tasks, and (2) lock x is free (i.e., the standard locking principle applies). The second condition is required just to avoid races *inside* a task; it can be dropped if tasks are single-threaded.

In [26,25], we have defined the iso-calculus, a typed call-by-value lambda calculus that is extended with threads, verlocks and tasks. It gives formal semantics to the constructs described above. The iso-calculus has a type system that *guarantees* that well-typed programs satisfy isolation. Programs that do not guarantee isolation are rejected. Formally, the following *type soundness* theorem holds: if expression e is typeable, then all terminating executions that evaluate ("compute") e to value v satisfy isolation. The type system essentially guarantees two properties: (1) all operations that need to be isolated are protected by verlocks (*"no race conditions can happen"*), and (2) all verlocks required by a task are known before the task commences (*"safe versioning"*). It builds on Flanagan and Abadi's [10] type system for *safe locking* with singleton kinds. The formal proof of the type soundness theorem for the iso-calculus appeared in [25].

Declarative Synchronization for Protocol Reuse. One of the promises of modular protocol design is the reuse of protocol components in different protocol stacks. However, in practice, protocol reuse is problematic. Concurrent components within a stack implement a synchronization policy. Reusing selected components in a different stack often requires to modify component code, so that it implements a policy of the new stack. Unfortunately, this is a counterexample to protocol modularity.

The above problem does not exist if we separate the synchronization and protocol code. Two approaches to such *separation of concerns* have been developed within our project: (1) *static* [24], in which synchronization policy (that may include isolation) is declared between components using *concurrency combinators*, and (2) *dynamic* [23], in which synchronization policy is declared between *semantic rôles* using abstract types. In [24], we defined a property, called *composition safety*, that informally means that any runtime execution of a protocol can satisfy the synchronization policy declared using the language of concurreny combinators. The main result of this work was to show that the property can be verified statically, thus eliminating runtime errors due to wrong composition.

3 Group Communication Stacks

Flexibility and modularity of a group communication middleware requires a protocol framework with the right features. However, this is not enough. Flexibility and modularity requires also to identify the right components (or protocols). In this section we first point out common features of the most representative group communication stacks implemented in the past. Then we describe the new architecture of our new *Fortika* group communication stack, and explain its advantages. A more detailed discussion can be found in [16].

3.1 Traditional Group Communication Stacks

In any group communication stack, *atomic broadcast* is one of the most fundamental communication primitives provided.[1] However, there is not one single way to implement atomic broadcast. The architecture of traditional group communication stacks is shown on Figure 2(a). Their main characteristics are the following:

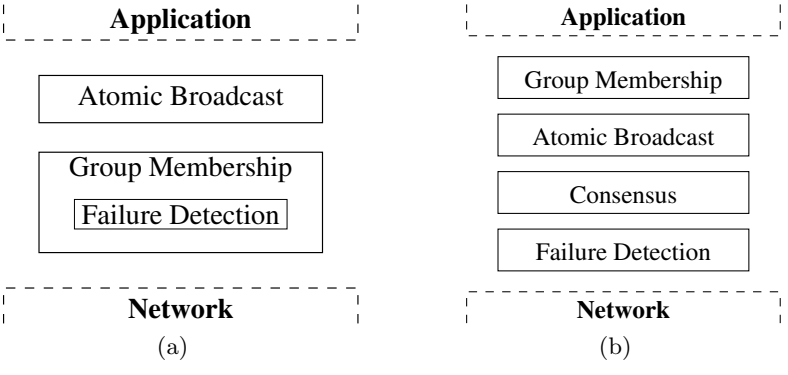

Fig. 2. (a) Traditional group communication stack, and (b) new group communication stack

Group membership and failure detection are strongly coupled: Failure detection is a low level mechanism that provides information (possibly incorrect) about the status *crashed*/*alive* of processes in a group (see Section 3.2). This information can be *inconsistent*, in the sense that two processes p, q might have a different perception of the processes in the group. On the contrary, the group membership service provides a *consistent* view of the successive membership of the group (see Section 4.1). In traditional architectures these two components are strongly coupled: failure detection is a sub-component of the group membership component, which acts as a failure detection for the rest of the system.

[1] *Atomic broadcast* delivers messages to all processes of a group in the same global order.

Atomic broadcast algorithms rely on group membership: The traditional protocol stacks use atomic broadcast algorithms that require the help of the group membership to avoid blocking in the case of the failure of some critical process. Basically these algorithms operate in two modes, a failure-free mode and a failure mode. A notification of removal of a process from the group (e.g., due to a failure suspicion) leads the protocol to switch to the failure mode.

The consensus abstraction is barely used: The traditional protocol stacks have not recognized the important role of consensus (see [21]) for solving agreement problems, e.g., group membership, atomic broadcast. These stacks have group membership (and not consensus) as their most basic component.

3.2 *Fortika*: The New Group Communication Stack

The architecture of Fortika, our new group communication stack, is shown on Figure 2(b). Fortika's protocols can be composed using several protocol frameworks: Samoa, Cactus or Appia. The main differences in Fortika's architecture compared with the traditional one are the following:

Group membership and failure detection are decoupled: The strong coupling between failure detection and group membership in the traditional stacks was motivated by the atomic broadcast algorithms (see Section 3.1). Decoupling group membership from failure detection has the following advantage: failure suspicions do not necessarily lead to the costly process exclusion operation.

Group membership relies on atomic broadcast and not the opposite: Atomic broadcast can be solved by a sequence of instances of consensus (see [21]). Such a solution does not rely on a group membership service, and works without blocking if no more than half of the processes in the group crash.

Since the group membership component does not need to be *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 the group membership service must deliver totally ordered views.

A consensus component is part of the stack: Since consensus plays a basic role in a group communication stack, it should appear as one of the bottom most component. Note that the consensus component requires the service of an (unreliable) failure detection component.

3.3 Assessment of the New Architecture

The two main advantages of the new architecture are the following.

Less complex stack: With traditional stacks, ordering is solved in two places: (1) within the group membership component for views, and (2) within the atomic broadcast component for messages. This is clearly not optimal, and introduces unnecessary complexity. The redundancy disappears in the new architecture.

Higher responsiveness: The performance of group communication must not only be measured in failure-free executions, but also in the case of failures. In the case of failures an important factor is the time needed to detect the failure (i.e., the crash of a process). The critical factor is thus the time-out of failure detectors. However, reducing the time-outs increases the probability of false suspicions. Decoupling failure suspicions from process exclusion has a big advantage: (1) a small value can be chosen for failure suspicion and (2) a large value for process exclusion. The factor that influences the performance of atomic broadcast is the short time-out value, and a wrong failure suspicion costs here very little. On the contrary, the exclusion of falsely suspected process has a high overhead, which can be avoided by choosing a large time-out value for exclusions.

4 Specifications

We discuss now more formal aspects related to group communication. We first address results related to the specification of group communication primitives. Here, we report on two contributions that each advance the state of the art, one by providing original specifications in the sparsely investigated territory of the crash-recovery model, and the other by revisiting and revising on specifications in the better investigated field of dynamic group communication. Finally, we propose an improved specification style for specifications, taking the example of unreliable failure detectors, to bring the respective mathematical definitions closer to our formal reasoning techniques.

4.1 Advances on Specifications in Refined Computing Models

Various models have been considered for group communication, namely *static/dynamic* groups, groups with *benign/malicious* faults, groups with *crash-stop/crash-recovery* processes [21]. These models not only influence the implementation of group communication, but also their specification. The model mostly considered in the literature is the static/crash-stop model with benign faults. Simple and widely adopted specifications for this model have been given in [12]. However, the static/crash-stop model does not cover the needs of a lot of applications. We discuss here group communication in the static crash-recovery model and in the dynamic crash-stop model.

Group Communication in the Crash-Recovery Model. In the crash-stop model processes do not have access to stable storage. In such a model a process that crashes loses all its state: upon recovery it cannot be distinguished from a newly starting process. The crash-stop model is attractive from an efficiency point of view: no stable storage means no costly logging operation, i.e., more efficient algorithms. However, the crash-stop model has also limitations. Algorithms developed in this model do not tolerate the crash of *all* processes. Moreover, access to stable storage is natural for many applications. This gives a strong motivation to consider group communication in the crash-recovery model. We consider here atomic broadcast.

Atomic broadcast in the crash-recovery model has been considered in [19]. However, as we explain in [15], the specification fails to capture the fundamental difference between the crash-stop and the crash-recovery model. In the crash-stop model there is no need to distinguish (1) the state of the application from (2) the state of the group communication infrastructure (if processes do not recover after a crash, the distinction is irrelevant). Indeed, in this case the two states are always trivially synchronized. This is no more the case when processes do recover. In this case the distinction—for each process—between *application state* and *group communication state*, requires to synchronize the checkpointing of these two parts of a process state. For this purpose we introduce a *commit* primitive. Thus atomic broadcast is defined in terms of the traditional *abcast* primitive (used by the application to broadcast a message), the traditional *adeliver* primitive (by which the group communication infrastructure provides a message to the application) and the new *commit* primitive. When executed by the application, *commit* tells the following to the group communication infrastructure: the application state, up to the most recent event, is saved on stable state. Implicitly, this leads the commit primitive to play two roles: (i) *veto* (e.g., no right to adeliver a message that was already adelivered before the commit), and (ii) *obligation* (e.g., for another process to adeliver a message).

Rather than giving here a formal specification of atomic broadcast in terms of *abcast*, *adeliver* and *commit*, we give the intuition of the specification on two examples.

(i) Veto role of commit. Consider the following two sequences of events on process p:

- *Scenario 1:* $adeliver_p(m)$; $crash_p$; $recovery_p$; $adeliver_p(m)$;
- *Scenario 2:* $adeliver_p(m)$; $\mathbf{commit}_p$; $crash_p$; $recovery_p$; $adeliver_p(m)$;

Scenario 1 is ok, but not Scenario 2. In Scenario 2, *commit* marks the point at which p's execution will resume after a crash. So m cannot be adelivered a second time after recovery. The absence of commit in the first scenario allows m to be adelivered again after p's recovery: the crash leads p to "forget" the adelivery of m.

(ii) Obligation role of commit. Consider the following two sequences of events on process p:

- *Scenario 1:* $adeliver_p(m)$; $crash_p$; $recover_p$;
- *Scenario 2:* $adeliver_p(m)$; $\mathbf{commit}_p$; $crash_p$; $recover_p$;

In Scenario 1, since p crashed after having adelivered m (i.e., the adelivery of m is "forgotten"), no other process is obliged to adeliver m. In Scenario 2, the execution of commit by p after the adelivery of m (i.e., upon recovery p "remembers" having adelivered m), forces other processes q to adeliver m. Note that the obligation is only on so-called *good* processes, i.e., processes that never crash or processes that crash only a finite number of times and always recover after a crash [5].

These two examples show that the execution of commit by process p makes all preceding events on p become "permanent". Without commit, events are

volatile. The specification of atomic broadcast in the crash-recovery model is based on this distinction. The details can be found in [15]. A prototype has been implemented in our *Fortika* group communication middleware (see Section 3).

Dynamic Group Communication. While the specification of group communication in the crash-recovery model has been addressed only by few authors, the specification of dynamic group communication has received a lot of attention [9]. Nevertheless these specifications are not really satisfactory. The main problem is that the specifications for dynamic groups are not close to the specifications for static groups. Specifically, if we consider the specifications for dynamic groups in the special case of a static group, we do not obtain the widely adopted static specifications [12].

In the existing group communication specifications for dynamic groups, the key component is the *group membership service*, which is responsible for adding and removing processes to/from a group. Consider some group g. The successive membership of g is modelled using the notion of *view*: the requirement on the group membership is that it delivers the successive views of g to its members in the *same order*. For example if $v_0(g) = \{p, q, r\}$ is the initial view of g, and then the successive views are $v_1(g) = \{p, q\}$ and $v_2(g) = \{p, q, s\}$, then all processes see the membership changes in the same order. In the existing group communication specifications for dynamic groups, the specification of the group communication service is used to specify the basic communication primitive, called *view synchronous broadcast* or simply *vscast*. Vscast basically requires that messages that are vscast are ordered with respect to view changes [9]. Finally, atomic broadcast is defined as vscast with an additional order property.

This might look similar to the specification of atomic broadcast with static groups, where atomic broadcast is defined as *reliable broadcast* (or *rbcast*) with an additional order property [21]. Unfortunately, when comparing the specifications of (i) rbcast with static groups and (ii) vscast with dynamic groups, it is hard to see their similarities. However, it is possible to specify dynamic group communication such that the dynamic specifications reduce to the standard static specifications when the group membership does not change.

With static groups, the specification distinguishes *correct* processes (that do not crash) and *faulty* processes (that crash). The obligations (to deliver messages) are only on correct processes. With dynamic groups, the situation is slightly different. Consider a group g. The obligations (to deliver messages) must only be on *correct processes that are members of* g. If some process crashes or leaves g, its obligation with respect to g disappear. Symmetrically, if a process joins g, it starts to have obligations with respect to g. This can be expressed by the notion of *g-correct* process, derived from the notion of *v-correct* process [22]. Informally, process p is *v-correct* in some view v if p installs view v and does not crash while its view is v; process p is *g-correct* if it is correct in the first view of g it belongs to, and in all successive views of g.

With the notion of *v-correct* and *g-correct* process we can define dynamic reliable broadcast almost as (static) reliable broadcast. Reliable broadcast is defined by validity, uniform agreement and uniform integrity (for a definition

of these properties, see [21]). Dynamic reliable broadcast can be defined by (i) the same uniform integrity property, (ii) slightly modified validity and uniform agreement properties (correct must be replaced with *g-correct* or *v-correct*), and (iii) a new *uniform same view delivery* property [22]:

– *Uniform same view delivery:* if two processes p and q deliver message m in view v^p (for p) and v^q (for q), then $v^p = v^q$.

This specification of dynamic reliable broadcast is a generalization of static reliable broadcast: if the group is static, dynamic reliable broadcast reduces to static reliable broadcast.

Dynamic atomic broadcast can then be defined as dynamic reliable broadcast with an additional total order property, which only slightly differs from the static total order property. Details can be found in [22], which also shows that the group membership specification can be trivially obtained from the dynamic atomic broadcast specification.

4.2 Proof-Oriented Specification Style for Failure Detectors

The concept of *unreliable failure detectors* was introduced by Chandra and Toueg [8] as a means to add weak forms of synchrony into asynchronous systems, mostly of the crash-stop model mentioned in the previous Section 4.1. Various kinds of such failure detectors, as we also use in the group communication architecture of Figure 2, have been identified as each being the weakest to solve some specific distributed programming problem [7]. Here, we provide—for the purpose of specification—a fresh look at the concept of failure detectors from the point of view of programming languages, using the formal tool of operational semantics, with the goal of bringing it closer to our formal reasoning techniques (see Section 5.1).

According to Chandra and Toueg [8], at any given time $t \in \mathbb{T}$, the failure detector (FD) of some process outputs a list of (names of) processes that it currently suspects to have crashed. As mentioned in Section 3, FDs are unreliable: they may make mistakes, they may disagree among themselves, and they may even change their mind indefinitely often.

In Table 1, we propose a uniform specification scheme—based on a two-layered transition system—to describe the operational semantics of process networks in the context of failure detectors. One layer describes—by separate sets of rules—both the transitions $N \longrightarrow N'$ of process networks (here, left unspecified to keep the setting parametric, but should be derived from the description of the algorithm) and the transitions $\Gamma \longrightarrow \Gamma'$ of the network's environment (keeping track of crashes and providing failure detection, as indicated by rule (ENV)). A process i in a network carries out essentially two kinds of transitions $N \longrightarrow N'$, distinguished by whether it requires the suspicion of some process j by process i, or not. Formally, we use labels $\text{suspect}_j@i$ and $\tau@i$ to indicate these two kinds. Another layer, with the rules (TAU) and (SUSPECT), deals exclusively with the compatibility of network and environment transitions, conveniently focusing on the environment conditions for the two kinds of transitions of process networks.

Table 1. Uniform "Abstract" Operational Semantics Scheme

$$(\textsc{env})\ \frac{\text{“ failure detection events happens in the environment ”}}{\Gamma\ \longrightarrow \Gamma'}$$

$$(\textsc{tau})\ \frac{\Gamma\ \longrightarrow \Gamma' \qquad N\ \xrightarrow{\tau@i} N' \qquad \text{“ } i \text{ not crashed in } \Gamma \text{ ”}}{\Gamma \vdash N\ \longrightarrow \Gamma' \vdash N'}$$

$$(\textsc{suspect})\ \frac{\Gamma\ \longrightarrow \Gamma' \qquad N\ \xrightarrow{\text{suspect}_j@i} N' \qquad \boxed{\text{“ } j \text{ may be suspected by } i \text{ in } \Gamma \text{ ”}} \qquad \text{“ } i \text{ not crashed in } \Gamma \text{ ”}}{\Gamma \vdash N\ \longrightarrow \Gamma' \vdash N'}$$

For example, the boxed condition exploits the failure detector information that in our scheme is to be provided via the environment component Γ.

Runs are sequences of system transitions, as derivable by operational semantics rules. A process is *correct in a given run*, if it does not crash in this run.

Chandra and Toueg specified FDs by means of *failure patterns* $F : \mathbb{T} \to 2^{\mathbb{P}}$ and *failure detector histories* $H : \mathbb{T} \times \mathbb{P} \to 2^{\mathbb{P}}$. We refer to the respective runs as $\mathbb{T}$-runs, since time $\mathbb{T}$ lies at the core of the statically fixed components F and H. This (F, H)-based model is easily reformulated in our two-layered scheme [17].

Probably the main novelty of Chandra and Toueg's paper [8] was the definition and study of a number of FDs that only differ in their degree of reliability, as expressed by a combination of safety and liveness properties. These are formulated in terms of permitted and enforced suspicions according to the respective failures reported in F and the failure detection recorded in H:

completeness addresses *crashed processes that must be suspected* by (the FDs of) "complete" processes.
accuracy addresses *correct processes that must not be suspected* by (the FDs of) "accurate" processes.

These properties are implicitly quantified for *all possible runs*. The words "complete" and "accurate" processes indicate some flexibility in the definition of the set of processes that the property shall be imposed on. Many instantiations of completeness and accuracy have been proposed.

Inspired by the FD called Ω [7], we observed that the common principle behind the (F, H)-based notions of accuracy is that of "justified trust". The key role is played by correct processes—those that, according to F, were *immortal* in the given run—that are *trusted forever* (according to H) in the given run, either eventually or already from the very beginning. In a *dynamic* operational semantics scenario, as opposed to the static view of (F, H), we rather model the moment when such a process becomes forever trusted. Dynamically, however, we must also ensure this process not to crash afterwards—it must become immortal at this very moment. We call such a process *trusted-immortal*.

Table 2. Operational Semantics Scheme with Reliable Information

$$(\mathbb{D}\text{-ENV})\ \frac{(\mathsf{TI} \cup TI) \cap C = \emptyset \qquad (\mathsf{C} \cup C) \cap TI = \emptyset \qquad |\mathsf{C} \cup C| \leq \mathrm{maxfail}(n)}{(\mathsf{TI}, \mathsf{C}) \quad \rightarrow \quad (\mathsf{TI} \uplus TI, \mathsf{C} \uplus C)}$$

$$(\mathbb{D}\text{-TAU})\ \frac{(\mathsf{TI}, \mathsf{C}) = \Gamma \ \rightarrow \Gamma' \qquad N \xrightarrow{\tau@i} N' \qquad i \notin \mathsf{C}}{\Gamma \vdash N \ \rightarrow \Gamma' \vdash N'}$$

$$(\mathcal{X}\text{-SUSPECT})\ \frac{(\mathsf{TI}, \mathsf{C}) = \Gamma \ \rightarrow \Gamma' \qquad N \xrightarrow{\mathrm{suspect}_j@i} N' \qquad i \notin \mathsf{C} \qquad \boxed{\mathrm{condition}_{\mathcal{X}}(\Gamma, j)}}{\Gamma \vdash N \ \rightarrow \Gamma' \vdash N'}$$

Note, here, that our treatment of trusted immortals is to be seen in the very same way as Chandra and Toueg's treatment of (F, H): the ultimate goal is to provide some mathematical device to specify (not: implement) in retrospective view "what may have happened" in an acceptable run according to FD-sensitive information. They fix this information statically, while we allow it to develop dynamically and, by that, we can simplify the style of specification.

With this idea, we proposed a new model [17] capable to represent *all* of the FDs of [8] solely based on information that is not fixed before a run starts, but is dynamically appearing along its way. It turns out that two kinds of information suffice: (1) which processes have crashed, and (2) which processes have become *trusted-immortal*. Both kinds of information may occur at any moment in time, but they remain irrevocable in any continuation of the current run.

In Table 2, environments $\Gamma = (\mathsf{TI}, \mathsf{C})$ record sets TI of trusted-immortal processes and sets C of crashed processes. Rule ($\mathbb{D}$-ENV) precisely models their nondeterministic appearance in full generality: in a single step, an environment may be increased by further trusted-immortal processes ($\in TI$) or crashed processes ($\in C$). Rule ($\mathbb{D}$-TAU) permits actions $\tau@i$ if $i \notin \mathsf{C}$. Rule ($\mathcal{X}$-SUSPECT) requires in addition that the suspected process j is permitted to be suspected by Γ, depends on the FD accuracy that we intend to model.

Our $\mathbb{D}$-representations of FDs are proved extensionally equivalent with the $\mathbb{T}$-representations proposed in [8] via mutual "inclusion" of their sets of runs. Essentially, this works by looking for a *mutual simulation* of $\mathbb{T}$-runs and $\mathbb{D}$-runs sharing the same network run (by projecting onto the N-component). In general, proofs using the new instead of the old representation are considerably simpler.

5 Verification and Testing

Verification and testing are complementary, but equally important aspects. Up to now, in our new architecture, we have focused on two aspects: the formal verification of the consensus component, based on an operational semantics, and the experimental validation of the Fortika group communication stack, through the technique of fault injection. While the formal verification seeks to prove that a consensus component does precisely what it should, the experimental

validation seeks to "stress-test" the robustness of a component by confronting with environment conditions that go beyond what it should be able to cope with.

5.1 Formal Verification of the Consensus Component

As pointed out in Section 3.2, consensus is a fundamental component that plays a basic role in the new architecture. The properties it provides to the above components are:

1. *Validity*: If a process decides a value v, then v was proposed by some process.
2. *Agreement*: No two correct processes decide differently.
3. *Termination*: Every correct process (eventually) decides some value.

Any correct implementation of the consensus component must guarantee the respect of these three properties. Since consensus is one of the basic building blocks of the new group communication stack (see Figure 2(b)), the correctness of its implementation is fundamental for the stability of the whole new architecture.

Many algorithms are available to implement the consensus component. This variety is due to the existence of different models (of communication, of failure, etc) and the necessity of taking advantage of the properties provided by underlying components. A common trait of many of these algorithms is that they are described in pseudo code—i.e., with neither formal syntax nor formal semantics—and the proofs of their correctness are given informally, with brief argumentations expressed in natural language. Thus, the pseudo code sometimes leaves space to interpretation and the correctness proofs sometimes require the readers to actually prove themselves substantial parts or subresults for which only informal arguments were given. To convincingly argue for the stability and correctness of the whole architecture, we consider it vital to rely on a consensus implementation that has been proved correct formally.

Given the model in which our architecture was placed (reliable communication, crash failure) and given the presence of failure detectors, the algorithm that we chose to implement the consensus component was the one proposed by Chandra and Toueg in [8]. This algorithm makes use of reliable broadcast and failure detector abstractions. It also assumes a majority of correct processes.

Before explaining the algorithm, it is worth here to briefly clarify the meaning of the terms Quasi-reliable Point-to-Point and Reliable Broadcast abstractions that we will use in our description. Both Quasi-reliable Point-to-Point and Reliable Broadcast are components that lie in what we call Network layer (Figure 2). As their name suggest, they are (quasi) reliable in that they guarantee the reception and non-corruption of messages sent by correct processes. Quasi-reliable Point-to-Point takes care of messages exchanged between two processes, while Reliable Broadcast distributes messages to all the correct processes in a group. Since the implementation of these two components does not influence the consensus algorithm, in our study we represent them as abstractions providing specific features and properties. Now everything is in place to describe the Chandra-Toueg consensus algorithm.

The Chandra-Toueg algorithm (Figure 3) proceeds in *rounds* and is based on the *rotating coordinator* paradigm: for each round number, a single process

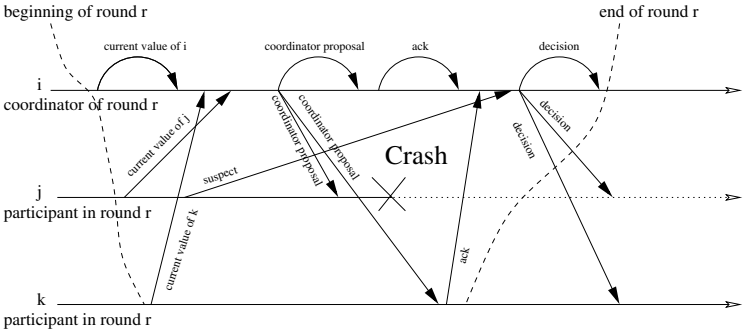

Fig. 3. A round in the Chandra-Toueg consensus algorithm

is predetermined to play a coordinator role, while all other processes in this round play the role of participants. Each of the n processes holds a local round counter and, at any time, knows who the coordinator of its current round is. For every round, each participant sends its current value to the coordinator of this round. The coordinator chooses one of the proposals it has received and sends it to all participants. These, in turn, are supposed to acknowledge the receipt. If the coordinator receives a majority of positive acknowledgments, it reliably broadcasts the value to all participants: this is going to be the decision. If a participant is not able to receive the coordinator proposal while waiting for it, and if the underlying failure detector component allows, then this participant may instead suspect the coordinator to have crashed. In this case, the participant sends a negative acknowledgement and moves to the next round, sending the very same value of the previous round to the new coordinator.

Correctness of the algorithm means that its resulting runs—more precisely: all runs satisfying the requirements on the underlying communication and coordination services—satisfy the three Consensus properties. In [8], Chandra and Toueg also provide sketches of proofs (written in natural language) of Validity, Agreement and Termination.

Such proofs make heavy reference to the concept of round. However, there is nothing like the global round number of a reachable global state in a system run. A round is only a local concept, and no relation between runs and asynchronous rounds is ever properly clarified. We consider this as problematic.

Moreover, there are two main reasons that make us argue against the approach of using only pseudo code. The first one is that the algorithm is not sufficiently complete. In fact, it describes only local behavior of processes and does not include any representation of the network. Also, the involved data structures are underspecified. In particular, there is almost no description of how and where the messages are buffered when waiting to be sent and once they have been received. The second reason is that, due to the absence of a precise formal semantics in the pseudo code, the algorithm description does not offer an unambiguous derivation of runs from the code. This is crucial because the specification of correctness properties is exclusively based on the notion of system runs. More precisely, it

is based on runs of the full distributed system, including the point-to-point and broadcast messages that are buffered within the network, as well as the behavior of the failure detector mechanism.

For these reasons, the proofs would profit much from the introduction of global knowledge on:

- system states and their past, which could provide us with precise information about which processes have been in which round in the past and what they precisely did when they were there;
- broadcast messages, which could provide us with precise information about what values are the chosen ones and about the processes that already know such values and the ones that still ignore them;
- point-to-point messages, which could provide us with precise information about what messages are ready to be sent but would still be lost in the case of a crash of the sender, what messages are about to be received and cannot be lost any longer should their original sender crash, what messages have already been received thus probably influencing the following behavior of their receiver.

Thus, in our work [18], we provide a mathematical structure that describes the global idealized run-time system comprising all processes and the network. Moreover, in order to simplify the proofs, our structure also plays the role of a system history, never forgetting any information during the computation.

Formally, the consensus algorithm is specified in terms of inference rules that define computation steps as transitions between configurations of the form:

$$\frac{\textit{some condition}}{\Gamma \vdash \langle \mathbf{B}, \mathbf{Q}, \mathbf{S} \rangle \rightarrow \Gamma' \vdash \langle \mathbf{B}', \mathbf{Q}', \mathbf{S}' \rangle}$$

Γ is the environment component, presented in Section 4.2, that records sets TI of trusted-immortal processes and sets C of crashed processes. $\mathbf{B}$ contains the (histories of) messages sent during the execution of the algorithm through the Reliable *B*roadcast service. $\mathbf{Q}$ contains the (histories of) messages sent during the execution of the algorithm through the *Q*uasi-reliable Point-to-Point service. $\mathbf{S}$ models the *state* of the n processes. We can access the current state of each single process using its identifier, i.e. $\mathbf{S}(i)$ is the current state of process i.

The condition for the execution of the rule is usually a condition on the state of a particular process and on the messages that such process has received up to that point. For example, "process i should be in a waiting state and the message containing the coordinator proposal should not figure among the messages received by i". The condition can also be extended with requirements on the environment component. For example, "the coordinator of round r should not be in the TI set". The execution of the rule modifies the contents of (some of) the structures. For example, the state of process i changes and a new message is sent, or broadcast.

As mentioned also in Section 4.2, runs are considered as sequences of system transitions derivable by operational semantics rules. In this way we can generate

all, and only, the runs that would be possible if we were executing the algorithm in reality. We can study such runs examining what happens step by step and verifying claims on the overall execution. The proofs of the properties of consensus are therefore made by showing that all the sequences of system transitions derivable by the given operational semantics rules, that represent all the possible runs generated by the algorithm, satisfy Validity, Agreement and Termination.

Summing up, in order to counter the observed incompleteness of the algorithm description in pseudo-code format, we have developed appropriate description techniques that incorporate the lacking information; also, in order to counter the observed ambiguity due to the lack of precise semantics of the pseudo-code with respect to the underlying system model, we have built the algorithm upon a formal description. We have then defined runs as sequential executions of the semantics rules and have kept track of the changes induced by the application of such rules in the **B**, **Q** and **S** structures. With this apparatus, we have eventually all the formal means for reasoning on rounds, as well as on time and on messages sent/in-transit/received, thus the proofs of correctness of the algorithm are now much more detailed, rigorous and credible.

5.2 Testing the Robustness of Fortika

Fault injection is a well-known technique to assess a system's resilience to error conditions. In a joint work with the University of Illinois [13], we set up an error-injection testbed in order to study how Fortika (see Section 3.2) reacts to data corruption. We carried out several error injection campaigns, which performed thousands of error injections, consisting in flipping a random bit in main memory or in a network message. It is important to point out that Fortika has been designed with a benign-fault model in mind, that is, only crash faults were considered (see Section 4.1). The memory and network corruption errors addressed in these experiments go far beyond the model's assumptions. Thus, our goals were not to find out whether Fortika is resilient to these faults, but rather to analize (and later minimize) any unacceptable behavior of the system.

Memory Injections. We performed a preliminary injection campaign for each memory segment: code (errors directly injected into the executed code), stack (local variables altered), and heap (allocated variables dynamically corrupted).

The most important result from these experiments was the high frequency of *partial process crashes*. In many experiments (26% for stack injections) the system completely hung. Further analysis showed that multithreading was behind such system-wide hangs: quite often, an error injection threw a Java runtime exception; the Java Virtual Machine stopped the offending thread, but let the others continue execution. We call this *partial crash*: some threads were working but some were not (e.g., the injected process could be sending heartbeats but omitting other messages). We enhanced the design of Fortika to cope with memory corruption and avoid partial crashes (as well as other problems, see details in [13]). Table 3 summarizes the results of memory injections for the improved Fortika design. This table shows the low rate of executions with unacceptable

behavior (between 0% and 6%) with respect to total manifested errors, which are defined as errors that visibly affect the system behavior (though not necessarily causing an incorrect execution).

Table 3. Memory injection results after improving Fortika

Memory Segment	Injected Errors	Manifested Errors	Unacceptable* Behavior
Heap	15177	1221	0
Text			
`libjava.so`	1000	616	36 (5.8%)
`libjvm.so`	910	269	7 (2.6%)
`libnet.so`	755	215	2 (1%)
Stack	5509	1825	109 (6.0%)

Table 4. Network injection results after improving Fortika

Total injected errors	1062
Manifested errors*	625
Message not detected	
a) No propagation	76 (12%)
b) Propagation	6 (1%)

* Percentages with respect to manifested errors are shown in parentheses.

Network Injections. In the network injection campaigns, an incoming message is altered (after checksum verification), thus resulting in an invalid input to the process. The goal is to analyze how far can an incorrect message get into the receiving process, and how badly it can affect the system. Most Fortika messages contain marshalled Java objects, thus, the desirable behavior is that Java unmarshalling routines detect and block incorrect messages.

A preliminary injection campaign evidenced a fairly high rate of incorrectly unmarshalled messages: up to 25% for certain messages types. In these experiments, the unmarshalling routines were unable to detect the corrupted message and allocated incorrect objects in memory. Further analysis showed that compatibility between different versions of the same Java class, a core feature of standard Java serialization, seriously harms robustness against corrupted messages. In the same way as with memory injections, we revised the design of Fortika in order to better react to incorrectly unmarshalled messages. The injection results for the new design are shown in Table 4, where we can see that only 13% of corrupted messages are not detected during unmarshalling, and sneack into the receiving process. Even in those cases, we see that the error seldom propagates to other processes (1% of all manifested errors).

6 Conclusion

We started from the goal to improve the state of the art of group communication by having groups with complementary scientific backgrounds—possibly characterized as distributed computing, programming languages and concurrency theory—join their forces. By now, we have already managed to achieve a number of interesting individual results that witness the potential for successful collaboration. Although much more work remains to be done, we consider our project as a promising first concrete step towards formally defined and verified implementations of flexibly reusable group communication middleware.

References

1. *The Appia project*. http://appia.di.fc.ul.pt/
2. The Cactus project. http://www.cs.arizona/Cactus/
3. The Ensemble project. http://www.cs.cornell/Info/Projects/Ensemble/
4. *The X-kernel project*. http://www.cs.arizona.edu/xkernel/
5. Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.
6. Daniel C. Bünzli, Sergio Mena, and Uwe Nestmann. Protocol composition frameworks, a header-driven model. In *Proceedings of the IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, USA, 2005.
7. Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of ACM*, 43(4):685–722, 1996.
8. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, 1996.
9. G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 4(33):1–43, December 2001.
10. Cormac Flanagan and Martin Abadi. Types for safe locking. In *Proc. ESOP '99*, LNCS 1576, March 1999.
11. Christophe Gensoul. Implementing Nuntius in the Objective Caml System. Master's thesis, EPFL, 2004.
12. V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. Technical Report 94-1425, Department of Computer Science, Cornell University, May 1994.
13. Sergio Mena, Claudio Basile, Zbigniew Kalbarczyk, André Schiper, and Ravi Iyer. Assessing the crash-failure assumption of group communication protocols. In *Proceedings of 16th IEEE Int'l Symp. on Software Reliability Engineering (ISSRE)*, November 2005.
14. Sergio Mena, Xavier Cuvellier, Christophe Grégoire, and André Schiper. Appia *vs.* Cactus: Comparing protocol composition frameworks. In *22nd Symposium on Reliable Distributed Systems. Florence, Italy*, October 2003.
15. Sergio Mena and André Schiper. A new look at atomic broadcast in the asynchronous crash-receovery model. In *Proceedings of the 24th Symposium on Reliable Distributed Systems (SRDS 2005)*, Orlando, Florida, October 2005.
16. Sergio Mena, André Schiper, and Paweł T. Wojciechowski. A Step Towards a New Generation of Group Communication Systems. In Markus Endler and Douglas Schmidt, editors, *Proceedings of Middleware 2003: The 4th ACM/IFIP/USENIX International Middleware Conference (Rio de Janeiro, Brazil)*, volume 2672 of *LNCS*, pages 414–432. Springer, June 2003.
17. Uwe Nestmann and Rachele Fuzzati. Unreliable failure detectors via operational semantics. In Vijay A. Saraswat, editor, *Proceedings of ASIAN 2003*, volume 2896 of *LNCS*, pages 54–71. Springer, December 2003.
18. Uwe Nestmann, Rachele Fuzzati, and Massimo Merro. Modeling consensus in a process calculus. In Roberto Amadio and Denis Lugiez, editors, *Proceedings of CONCUR 2003*, volume 2761 of *LNCS*, pages 399–414. Springer, August 2003.
19. L. Rodrigues and M. Raynal. Atomic Broadcast in Asynchronous Crash-Recovery Distributed Systems and Its Use in Quorum-Based Replication. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1205–1217, September 2003.
20. *The SAMOA Protocol Framework*. http://lsrwww.epfl.ch/samoa.
21. A. Schiper. Dependable Systems. This book, Part I, Chapter 2.

22. A. Schiper. Dynamic Group Communication. *ACM Distributed Computing*, 18(5):359–374, April 2006.
23. Vlad Tanasescu and Paweł T. Wojciechowski. Role-based declarative synchronization for reconfigurable systems. In Manuel Hermenegildo and Daniel Cabeza, editors, *Proceedings of PADL 2005: The 7th International Symposium on Practical Aspects of Declarative Languages (Long Beach, CA, USA)*, volume 3350 of *LNCS*, pages 52–66. Springer, January 2005.
24. Paweł T. Wojciechowski. Concurrency combinators for declarative synchronization. In Wei-Ngan Chin, editor, *Proceedings of APLAS 2004: The 2nd Asian Symposium on Programming Languages and Systems (Taipei, Taiwan)*, volume 3302 of *LNCS*, pages 163–178. Springer, November 2004.
25. Paweł T. Wojciechowski. Isolation-only transactions by typing and versioning. Technical Report IC-2004-104, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL), December 2004. 47pp.
26. Paweł T. Wojciechowski. Isolation-only transactions by typing and versioning. In *Proceedings of PPDP '05: The 7th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (Lisboa, Portugal)*, July 2005.
27. Paweł T. Wojciechowski and Olivier Rütti. On correctness of dynamic protocol update. In *Proceedings of FMOODS '05: The 7th IFIP Conference on Formal Methods for Open Object-Based Distributed Systems (Athens, Greece)*, volume 3535 of *LNCS*, pages 275–289. Springer, June 2005.
28. Paweł T. Wojciechowski, Olivier Rütti, and André Schiper. SAMOA: Framework for Synchronisation Augmented Microprotocol Approach. In *Proceedings of IPDPS 2004: The 18th IEEE International Parallel and Distributed Processing Symposium (Santa Fe, USA)*, April 2004.