Ruminations on Domain-Based Reliable Broadcast

Svend Frølund   Fernando Pedone
Hewlett-Packard Laboratories
Palo Alto, CA 94304, USA

Abstract

A distributed system is no longer confined to a single administrative domain. Peer-to-peer applications and business-to-business e-commerce systems, for example, typically span multiple local-area and wide-area networks, raising issues of trust, security, and anonymity. This paper introduces a distributed systems model with an explicit notion of *domain* that defines the scope of trust and local communication within a system. We introduce leader-election oracles that distinguish between common and distinct domains, encapsulating failure-detection information and leading to modular solutions and proofs. We show how Reliable Broadcast can be implemented in our domain-based model, we analyze the cost of communicating across groups, and we establish lower-bounds on the number of cross-domain messages necessary to implement Reliable Broadcast.

1 Introduction

1.1 Motivation

Distributed systems are no longer confined to a single administrative domain. For example, peer-to-peer applications and business-to-business e-commerce systems typically span multiple local-area and wide-area networks. In addition, these systems commonly span multiple organizations, which means that issues of trust, security, and anonymity have to be addressed. Such global environments present new challenges to the way we program and organize distributed systems.

At the programming level, several researchers have recently proposed constructs to decompose a global system into smaller units that define boundaries of trust and capture locality. An example of such a construct is the notion of ambient introduced by Cardelli [Car99b]. The trend reflects a growing realization that one should not treat a global system as a very large local-area network.

1.2 The Domain-Based Model

We introduce a distributed systems model with an explicit notion of *domain* that defines the scope of trust and local communication within a system. A domain is a set of processes, and a system is a set of domains. A domain-based model allows us to employ novel complexity measures, such as the number of times a given algorithm communicates across domain boundaries. Such complexity measures reflect the real-world costs of crossing firewalls and communicating along wide-area links. In contrast, a conventional "flat" model, with a single set of processes, attributes the same cost to any point-to-point communication. Besides complexity measures, a domain-based model also allows us to capture the realistic notion that failure information within a local-area network is more accurate than failure information obtained across wide-area networks. We develop a notion of leader-election oracle that provides one level of information to processes within the same domain, and another, with weaker properties, to processes in other domains. Finally, domains enable us to reflect the common security policy that a process trusts other processes in the same organization (domain), but not necessarily processes in other domains. We introduce Byzantine failures into our model, and having domains allows us to attribute Byzantine behavior to entire domains (as seen from the outside).

1.3 Domain-Based Algorithms

In building global systems, a fundamental concern is the reliable dissemination of information. We want the information dissemination to be reliable, but also efficient and scalable. In practice, an important aspect of efficiency is the amount of wide-area bandwidth consumed by information dissemination algorithms. In terms of scalability, an important quality is to make the system decentralized. To address the issue of information dissemination in large-scale systems, we describe a number of Reliable Broadcast algorithms that work in our domain-based model.

The first set of algorithms tolerate crash failures only. In this context, we start by examining how to implement Reliable Broadcast in a purely asynchronous, domain-based model. We then incrementally add synchrony assumptions to this model, and show how one can exploit these assumptions, in the form of leader-election oracles, to reduce the number of messages that cross domain boundaries. We analyze the cost of these algorithms in terms of cross-domain messages and present lower bounds that quantify the inherent cost of reliability in a domain-based model.

The second set of algorithms implement Reliable Broadcast in a system with Byzantine failures. We consider a domain to be Byzantine if it contains a Byzantine process, and we develop algorithms that can tolerate Byzantine domains. Considering Byzantine behavior at the domain level reflects the notion that a domain is the unit of trust and security: once a domain has been compromised, it is likely that an adversary can take over as many processes as it wishes within that domain. We first provide a protocol that ensures agreement: all correct processes in all non-Byzantine domains deliver the same set of messages. We then give an algorithm that also ensures *consistency:* messages are tagged with unique identifiers, and no two correct processes in non-Byzantine domains deliver different messages with the same identifier. Consistency is an important property. It prevents spurious messages from being delivered, which may happen, for example, if an erroneous sender keeps using the same message identifier with different message contents.

1.4 Related Work

Compared with the ambient calculus [Car99a], and other wide-area models based on process algebras, our system model explicitly captures failures and the availability of failure information.

A number of papers have addressed the issue of information dissemination with Reliable Broadcast. The algorithms in [HT93] use message diffusion, and tolerate crash and link failures. In [CT90], the authors present time and message efficient Reliable Broadcast algorithms that tolerate crash and omission failures. The protocol in [GS96] exploits failure detection to more efficiently diffuse messages. All these protocols assume a flat universe of processes. If we were to employ them on top of a networking infrastructure with wide-area networks, the resulting wide-area message complexity would be proportional to the total number of processes. With our protocols, the wide-area message complexity is proportional to the number of domains (assuming that there are no wide-area links within a domain).

The Reliable Broadcast algorithm in [Rei94] assumes a flat model with Byzantine processes. In our terminology, the algorithm implements agreement but not consistency—there is no notion of message identifiers to ensure that processes deliver the same message content for the same identifier. The notion of Byzantine Agreement [PSL80, LSP82] captures a variation of Reliable Broadcast in the Byzantine model. With Byzantine Agreement, a single process broadcasts a value, and all correct processes must decide on the same value. The original formulation of the problem in [PSL80] requires an explicit notion of time. The definition of Asynchronous Byzantine Agreement [BT85] does not use time and only requires honest processes to decide if the broadcasting process is honest. Asynchronous Byzantine Agreement ensures consistency relative to a single message, but the definition of the problem is for a single message. Thus, besides implementing Byzantine-tolerant Reliable Broadcast in a domain-based model, our algorithms also bridge the gap between Reliable Broadcast and Byzantine Agreement.

1.5 Summary of Contributions

The paper makes the following contributions:

- We define a domain-based system model, which corresponds to current wide-area distributed systems, and give a specification of Reliable Broadcast in this model.
- We introduce leader-election oracles that distinguish between common and distinct domains, encapsulate failure-detection information, and lead to modular solutions and proofs.

- We show how Reliable Broadcast can be implemented in our domain-based model. We start with the simple case of only two groups and then build up to more complex cases.
- We analyze the cost of communicating across groups, evaluate the performance of our protocols in terms of cross-domain messages, and provide lower-bounds on the number of cross-domain messages necessary to implement Reliable Broadcast.

2 System Model and Definitions

2.1 Processes, Failures and Communication

We assume that the system is composed of groups of processes, that is, $\Pi = \{\Pi_1, \Pi_2, ..., \Pi_n\}$, where $\Pi_x = \{p_1, p_2, ..., p_{n_x}\}$. When we need to distinguish processes from different groups, we will use superscripts: $p_i^x \in \Pi_x$. Processes may be *honest* (i.e., they execute according to their protocols) or *malicious* (i.e., Byzantine). Honest processes can crash, but before they crash, they follow their protocols.

A process that is honest and does not crash is *correct*; if the process is honest but crashes it is *faulty.* If a group has at least one malicious process, then it is *bad*; otherwise the group is *good.* Therefore, good groups can contain only correct and faulty processes, while bad groups can contain any kind of processes. In a good group Π_x, we assume that at most $f_x < n_x$ processes crash.

Processes communicate by message passing. Each message m has three fields: $sender(m)$, the process where m originated, $id(m)$, a unique identifier associated with m, and $val(m)$, the actual contents of m. We assume that the network is fully connected, and each link is reliable. A reliable link guarantees that (a) if p_i sends a message m to p_j, and both p_i are p_j are correct, then p_j eventually receives m; (b) each message is received at most once by honest processes; and (c) if an honest process receives a message m, and if sender(m) is honest, then m was sent.

The system is asynchronous: message-delivery times are un-bounded, as is the time it takes for a process to execute steps of its local algorithm. We assume the existence of a discrete global clock, although processes do not have access to it—the global clock is used only to simplify some definitions. We take the range $\mathcal{T}$ of the clock's ticks to be the set of natural numbers.

2.2 Domain-Based Reliable Broadcast

The domain-based Reliable Broadcast abstraction is defined by the primitives byz-broadcast(m) and byz-deliver(m), and has the following properties:

- *(Validity.)* If a correct process in a good group byz-broadcasts m, then it byz-delivers m.
- *(Agreement.)* If a correct process in a good group byz-delivers m, then each correct process in every good group also byz-delivers m.
- *(Integrity.)* Each honest process byz-delivers every message at most once. Moreover, if sender(m) is honest, then sender(m) byz-broadcast m.
- *(Consistency.)* Let p_i and p_j be two processes in good groups. If p_i byz-delivers m, p_j byz-delivers m', and $id(m) = id(m')$, then $val(m) = val(m')$.

Domain-based Reliable Broadcast without consistency is a generalization of Reliable Broadcast in a flat model. Throughout the paper, we use Reliable Broadcast locally in groups as a building block to implement domain-based Reliable Broadcast. We refer to such an abstraction as local Reliable Broadcast. Local Reliable Broadcast is defined by the primitives r-broadcast(m) and r-deliver(m). The properties of local Reliable Broadcast can be obtained from the properties of domain-based Reliable Broadcast by replacing byz-broadcast(m) and byz-deliver(m) by r-broadcast(m) and r-deliver(m), and considering a system composed of one good group only. Without the consistency property, local Reliable Broadcast can implemented with a conventional "flat" algorithm [CT96]. When the consistency property is needed, such as in our most general algorithm in Section 5.2, the implementation is different from Reliable Broadcast implementations in the flat model. We discuss such an implementation further in Section 5.2.

2.3 Leader-Election Oracles

In some of our algorithms we use oracles that give hints about process crashes—they do not provide any information about which processes are malicious. Our oracles are quite similar to the Ω failure detector in [CHT96]. Where Ω is defined for a "flat" system, our oracles are defined for a distributed system with groups.

We introduce a notion of group oracle—an oracle that gives information about the processes in a particular group. For example, the group oracle Ω_x gives information about the processes in Π_x. Thus, our system contains a set of oracles, $\{\Omega_1, \Omega_2, \ldots, \Omega_n\}$, one per group. Each process has access to all oracles. Having an oracle per group, rather than a single "global" oracle, allows us to distinguish between local information and remote information. A process p_i^x gets local information from the oracle Ω_x (information about other processes in Π_x). In contrast, a process p_i^x obtains remote information from an oracle Ω_y, where $y \neq x$ (information about processes in other groups). We use the notion of group oracle to model a system where local information is stronger than remote information.

We use the set G to denote the set of all processes in the system ($G = \Pi_1 \cup \Pi_2 \cup \ldots \cup \Pi_n$). Moreover, we use the set good to denote the set of good groups ($\mathsf{good} \subseteq \Pi$).

In the following, we adapt the model in [CHT96] to define a notion of group oracle. A failure pattern represents a run of the system. A failure pattern F captures which processes in G crash, and when they crash. Formally speaking, a failure pattern is a map from time to a subset of G. Based on a failure pattern F, we can define the set of processes that crash in F as $\mathsf{crash}(F)$:

$$F \in \mathcal{F} = \mathcal{T} \rightarrow 2^G \quad (1)$$

$$\mathsf{crash}(F) = \cup_{t \in \mathcal{T}} F(t) \quad (2)$$

$$\mathsf{correct}_x(F) = \Pi_x \setminus \mathsf{crash}(F), \quad \text{if } \Pi_x \in \mathsf{good} \quad (3)$$

where $\mathcal{F}$ is the set of all failure patterns, and F is an element of this set. For a good group Π_x, the set $\mathsf{correct}_x(F)$ is the set of correct processes in Π_x.

A group-oracle history H_x is a map from process-time pairs to a process in Π_x.[1] A pair (q, t) maps to the process p_i^x if the process q at time t believes that p_i^x has not crashed. We also say

[1]The concept of a group-oracle history is similar to the notion of failure-detector history in [CHT96]. Where a failure-detector history is global, a group-oracle history is local to a particular group. Furthermore, where a failure-detector history maps to a set of processes (the processes that have failed at a given time), a group-oracle history maps to a single process (a process that is believed not to have crashed).

that q *trusts* p_i^x at time t. Intuitively, a failure pattern is what actually happens in a run, and a group-oracle history represents the output from a group oracle. We can now define a group oracle Ω_x as a map from a failure pattern to a set of group-oracle histories:

$$H_x \in \mathcal{H}_x = (G \times \mathcal{T}) \rightarrow \Pi_x \quad (4)$$

$$\Omega_x \in \mathcal{D}_x = \mathcal{F} \rightarrow 2^{\mathcal{H}_x} \quad (5)$$

The set $\mathcal{H}_x$ is the set of all group-oracle histories relative to a group Π_x, and the history H_x is an element in this set. Furthermore, the set $\mathcal{D}_x$ is the set of all oracles for Π_x, and Ω_x is an element in this set (in other words, Ω_x is an oracle).

We can use the above definitions to establish constraints on the information that an oracle Ω_x gives a process p. First of all, if Π_x is a bad group, there are no constraints—Ω_x may return arbitrary information. If Π_x is a good group, there are two sets of constraints: a set of constraints for local information ($p \in \Pi_x$) and remote information ($p \notin \Pi_x$):

- *Local Trust:* For any good group Π_x, eventually all correct processes in Π_x trust the same correct process in Π_x. Formally:

$$\Pi_x \in \mathsf{good} \Rightarrow \big\langle \forall F, \forall H_x \in \Omega_x(F), \exists t \in \mathcal{T}, \exists q \in \mathsf{correct}_x(F), \forall p \in \mathsf{correct}_x(F), \forall t' \geq t : H_x(p, t') = q \big\rangle \quad (6)$$

- *Remote Trust:* For any good group Π_x, eventually, all correct processes in all good groups trust a correct process in Π_x. Formally:

$$\Pi_x \in \mathsf{good} \Rightarrow \big\langle \forall F, \forall H_x \in \Omega_x(F), \exists t \in \mathcal{T}, \forall \Pi_y \in \mathsf{good}, \forall p \in \mathsf{correct}_y(F), \forall t' \geq t : H_x(p, t') \in \mathsf{correct}_x(F) \big\rangle \quad (7)$$

- *Stability:* For any good group Π_x, eventually, any correct process in a good group trusts the same process in Π_x forever. Formally:

$$\Pi_x \in \mathsf{good} \Rightarrow \big\langle \forall F, \forall H_x \in \Omega_x(F), \exists t \in \mathcal{T}, \forall \Pi_y \in \mathsf{good}, \forall p \in \mathsf{correct}_y(F), \exists q \in \mathsf{correct}_x(F), \forall t' \geq t : H_x(p, t') = q \big\rangle \quad (8)$$

Roughly speaking, a group oracle Ω_x is equivalent to the oracle Ω in [CHT96] in terms of local information—we use a group oracle Ω_x for leader election within the group Π_x. In terms of remote information, Ω_x provides slightly weaker information—the processes in a group Π_y use the oracle Ω_x to select a process in Π_x that serves as destination for inter-group messages that are sent from Π_y. The *Remote Trust* and *Stability* properties ensure that any process in a remote group eventually trust a single process in Π_x. However, different processes in remote groups may trust different processes in Π_x. If instead the oracle Ω_x guaranteed to eventually return the same process in Π_x to any process (local and remote), this "leader" could become a bottleneck since it would handle all incoming and outgoing communication in Π_x.

3 Abstractions for Solving Reliable Broadcast

It is possible to implement domain-based Reliable Broadcast in a purely asynchronous system. However, one can come up with more efficient algorithms in a model with oracles. We want to provide insights about both types of algorithms: algorithms that assume a purely asynchronous model and algorithms that use the oracles introduced in Section 2.3. It turns out that both types of algorithms share a common principle: for each broadcast message, at least one correct process in the sending group communicates the message to a correct process in the receiving group. The choice of underlying model does not change this principle, only how it is achieved. Rather than describe a number of algorithms that are identical except for their dealing with the underlying model, we encapsulate model-related concerns in well-defined abstractions. The use of these abstractions allows us to simplify the description of our algorithms and to modularize the proof of their correctness.

Our abstractions are specified relative to a given group. Rather than explicitly pass a group as parameter to every invocation, we supply the group as a subscript of the abstraction. For example, the abstraction $\mathsf{senders}_x()$ means "the senders abstraction relative to a group Π_x."

The $\mathsf{senders}_x()$ abstraction. This abstraction is used within a group Π_x to select the processes in Π_x that send messages to processes in other groups. The $\mathsf{senders}_x()$ abstraction returns a set of processes in Π_x. If p_i^x in Π_x invokes $\mathsf{senders}_x()$ and the returned set includes p_j^x, we say that p_i^x *selects* p_j^x. The $\mathsf{senders}_x()$ abstraction has the following properties:

- *Termination:* The $\mathsf{senders}_x()$ abstraction is non-blocking.
- *Validity:* Eventually, $\mathsf{senders}_x()$ selects a correct process in Π_x.
- *Agreement:* Eventually, any invocation of $\mathsf{senders}_x()$ selects the same processes.

Notice that the $\mathsf{senders}_x()$ abstraction is only available to processes in the group Π_x. Notice also that the set of processes returned by $\mathsf{senders}_x()$ may change over time.

The $\mathsf{destinations}_x()$ abstraction. Processes in a group Π_y can use the $\mathsf{destinations}_x()$ abstraction to select the recipients in Π_x of inter-group messages. That is, processes in Π_y use $\mathsf{destinations}_x()$ to determine which processes in Π_x to send messages to. If a process in Π_y, p_i^y, invokes $\mathsf{destinations}_x()$, and if the returned set includes a process p_j^x, we say that p_i^y *selects* p_j^x. The $\mathsf{destinations}_x()$ abstraction has the following properties:

- *Termination:* The $\mathsf{destinations}_x()$ abstraction is non-blocking.
- *Validity:* Eventually, $\mathsf{destinations}_x()$ selects a correct process in Π_x.
- *Agreement:* For any process p, eventually any invocation of $\mathsf{destinations}_x()$ by p selects the same processes.

Notice that $\mathsf{destinations}_x()$ is a global abstraction—any process in any group can invoke this abstraction under the above guarantees. Although the properties of $\mathsf{destinations}_x()$ and $\mathsf{senders}_x()$ are quite similar, the Agreement properties are different. For $\mathsf{senders}_x()$, the Agreement property ensures that all processes in Π_x eventually select the same set of senders. For $\mathsf{destinations}_x()$, the Agreement property only ensures that any individual process "stabilizes" on the same set of destinations, different processes may stabilize on different sets of destinations. The weaker Agreement property for $\mathsf{destinations}_x()$ implies that processes within Π_x can share the load: the abstractions do not insist that only a single process receives all messages in Π_x.

Implementing the $\mathsf{senders}_x()$ **and** $\mathsf{destinations}_x()$ **abstractions.** In an asynchronous model, we can implement the abstractions by returning a subset of Π_x that contains at least $f_x + 1$ processes (such a set will contain at least one correct process). In a model with oracles, we can implement the abstractions by simply returning the single process output from the oracle. We show these implementations in Table 1.

	asynchronous implementation	oracle-based implementation
$\mathsf{senders}_x()$, $\mathsf{destinations}_x()$	**return** $\{p_j^x \mid j \le f_x + 1\}$	**return** Ω_x

Table 1: Implementation of the abstractions with or without oracles

In the asynchronous implementation, we return the same set of processes in both abstractions. However, there is no requirement to do so: the implementation of $\mathsf{senders}_x()$ could return one subset of size $f_x + 1$ and the implementation of $\mathsf{destinations}_x()$ could return another. Moreover, although the oracle-based implementation of $\mathsf{senders}_x()$ is identical to the implementation of $\mathsf{destinations}_x()$, the abstractions still provide different agreement properties: $\mathsf{senders}_x()$ is only called by processes in Π_x, and the oracle gives stronger guarantees to processes within Π_x.

The $\mathsf{send}_x()$ **and** $\mathsf{receive}_x()$ **abstractions.** The $\mathsf{send}_x()$ and $\mathsf{receive}_x()$ abstractions capture reliable communication between groups. The $\mathsf{send}_x()$ abstraction takes a message as argument, and the $\mathsf{receive}_x()$ abstraction returns a message. The two abstractions have the following properties:

- *Termination:* The $\mathsf{send}_x()$ abstraction is non-blocking.
- *Validity:* If a correct process in Π_y invokes $\mathsf{send}_x()$ with a message m then eventually a correct process in Π_x can receive m by invoking $\mathsf{receive}_x()$.
- *Integrity:* If $\mathsf{receive}_x()$ returns a message m to an honest process p, and if sender(m) is honest, then sender(m) called $\mathsf{send}_x()$ with m.

Unlike a traditional message-sending operation, $\mathsf{send}_x()$ takes a group, not a process, as the message destination—we use a subscript to designate the group. The $\mathsf{send}_x()$ abstraction encapsulates the concern of sending to a correct process. This is in contrast to $\mathsf{destinations}_x()$, which simply ensures that some correct process is selected.

We can implement $\mathsf{send}_x()$ with regular point-to-point communication primitives and the $\mathsf{destinations}_x()$ abstraction previously introduced (see Algorithm 1).

4 A Special Case: Two Good Groups

4.1 The Algorithm

We examine how to implement domain-based Reliable Broadcast. For simplicity, we restrict our scope to systems with only two good groups—our algorithm tolerates crash failures only. We introduce algorithms that work for any number of groups and tolerate malicious processes in Section 5. Throughout this section, we consider two good groups, Π_x and Π_y, and assume that messages originate in Π_x.

Algorithm 1 Implementation of $\mathsf{send}_x()$ and $\mathsf{receive}_x()$ based on destination selection

procedure $\mathsf{send}_x(m)$
  $dest \leftarrow \mathsf{destinations}_x()$
  send [GS,m] to all processes in $dest$
  fork task watch(m,$dest$,x)

task watch(m,$dest$,x)
  while TRUE
    if $dest \neq \mathsf{destinations}_x()$ **then**
      $dest \leftarrow \mathsf{destinations}_x()$
      send [GS,m] to all processes in $dest$

when receive([GS,m])
  $\mathsf{receive}_x(m)$

We are interested in algorithms that are judicious about the number of inter-group messages used to deliver a broadcast message in both groups. We present a generic algorithm that relies on the abstractions in Section 3. By instantiating the abstractions in different ways (with or without using oracles), we achieve solutions for different models.

Algorithm 2 has four *when* clauses, but only one executes at a time. Whenever one of the *when* conditions evaluates true, the clause is executed until the end. If more than one condition evaluates true at the same time, one clause is arbitrarily chosen.

Algorithm 2 Reliable broadcast for two good groups

Initially:
  $rcvMsgs \leftarrow \emptyset$
  $fwdMsgs \leftarrow \emptyset$

procedure byz-broadcast(m)
  r-broadcast($\{m\}$)

when r-deliver($mset$)
  for all $m \in mset \setminus rcvMsgs$ **do** byz-deliver(m)
  $rcvMsgs \leftarrow rcvMsgs \cup mset$

when $p_i \in \mathsf{senders}_x()$ **and** $rcvMsgs \setminus fwdMsgs \neq \emptyset$
  for all $m \in rcvMsgs \setminus fwdMsgs$ **do** $\mathsf{send}_y([\mathrm{FW}, m])$
  $fwdMsgs \leftarrow rcvMsgs$

when $\mathsf{receive}_y$([FW,m])
  send [LC,m] to all processes in Π_y

when receive [LC,m] for the first time
  byz-deliver(m)

4.2 Algorithm Assessment

Because Algorithm 2 only relies on the specification of our abstractions, and not on their implementation, it is possible to mix and match abstractions with different implementations. For example, it is possible to combine an oracle-based implementation of senders() with an asynchronous implementation of destinations(). Such a combination would exploit synchrony assumptions within groups but not across groups.

The various combinations of abstraction implementations give rise to different costs in inter-group messages for the resulting Reliable Broadcast algorithm. If we use the asynchronous implementation of both abstractions in Algorithm 2, we achieve a performance of $(f_x+1)(f_y+1)$ inter-group messages per broadcast.

If we combine the asynchronous implementation of $\mathsf{destinations}_y()$ with an oracle-based implementation of $\mathsf{senders}_x()$, we obtain a domain-based Reliable Broadcast algorithm with a best-case message cost of $f_y + 1$. The algorithm may have a higher message cost for arbitrary periods of time, but the properties of Ω_x ensure that eventually only a single process in Π_x will be selected. Thus, eventually only a single process will send inter-group messages. Moreover, with the asynchronous implementation of $\mathsf{destinations}_y()$, the number of destinations is constant (i.e., $f_y + 1$), and so is the number of messages sent by each selected sender.

If instead we combine the asynchronous implementation of $\mathsf{senders}_x()$ with an oracle-based implementation of $\mathsf{destinations}_y()$, we obtain a domain-based Reliable Broadcast algorithm with best-case message cost of f_x+1. With an asynchronous implementation of $\mathsf{senders}_x()$, there will always be $f_x + 1$ senders. Due to the stability property of the oracle Ω_y, each of the $f_x + 1$ senders will eventually trust a correct process in Π_y forever. Thus, at any of the $f_x + 1$ senders there is a time t after which $\mathsf{destinations}_y()$ returns the same process forever. This means that after t, the watch task in Algorithm 1 does not send any messages. Thus, after t, each broadcast message results in each of the $f_x + 1$ senders sending exactly one inter-group message. Notice that the senders do not necessarily send to the same process in Π_y.

Finally, if we combine the oracle-based implementation of $\mathsf{senders}_x()$ with the oracle-based implementation of $\mathsf{destinations}_y()$, the resulting Reliable Broadcast algorithm will have best case of 1 inter-group message per broadcast. As we discussed above, there is a time after which the oracle Ω_x results in the selection of the single sender in Π_x. Furthermore, there is a time after which the Ω_y oracle returns the same destination forever to each sender. In combination, the two oracles ensure that, eventually, each broadcast message results in only a single process in Π_x sending a single message to a single process in Π_y.

4.3 Some Lower Bounds

It is easy to see that when both $\mathsf{senders}_x()$ and $\mathsf{destinations}_y()$ use oracle-based implementations, our resulting algorithm has an optimal best-case cost in terms of inter-group messages: no algorithm can solve Reliable Broadcast without exchanging at least one message between groups.

We informally argue now that if either $\mathsf{senders}_x()$ or $\mathsf{destinations}_y()$ (but not both) has an oracle-based implementation, our resulting algorithms also have optimal best-case costs. The argument is similar for both situations, and so, assume that $\mathsf{senders}_x()$ has an implementation that uses oracles. What we want to show is that the inter-group message cost of our algorithm, namely f_y+1, is a lower bound for algorithms where the sending group does not have information about failures in the receiving group. In the best case, a correct process p_i in Π_x is selected to send messages to processes in Π_y. Assume for a contradiction that p_i sends only f_y messages.

Consider a run where each process in Π_y that receives a message fails right after receiving the message; the remaining processes in Π_y will then never receive the message, and therefore, cannot deliver it.

In a purely asynchronous system (when neither the implementation of $\mathsf{senders}_x()$ nor the implementation of $\mathsf{destinations}_y()$ use oracles), our algorithm has inter-group message cost of $(f_x+1)(f_y+1)$, which is not optimal. Consider, for example, the special case when both n_x and n_y are greater than $f_x + f_y$. In this case, the following algorithms solves domain-based Reliable Broadcast: the first f_x+f_y+1 processes in Π_x send one message to the first f_x+f_y+1 processes in Π_y. The resulting message cost is $f_x + f_y + 1$. Moreover, from Proposition 1, it turns out that this algorithm is optimal.

Proposition 1 *Let $\mathcal{A}$ be a reliable broadcast algorithm in which processes do not query any oracle. For every run of $\mathcal{A}$ in which a correct process in Π_x byz-broadcasts some message, at least $f_x + f_y + 1$ messages are sent from Π_x to Π_y.*

PROOF: The proof is by contradiction. Assume that processes in Π_x send only $f_x + f_y$ messages to Π_y. Let R be a failure-free run in which p_i in Π_x byz-broadcasts message m and S the set of processes that send messages to Π_y in R.

We claim that there exists a run R' in which p_i also byz-broadcasts m, each process in $S_x \subseteq S$ crashes, and no process sends more messages in R' than it sends in R. Let p_j be a process in S_x that crashes at time t_j and p_k some process that sends one more message in R' at time $t > t_j$. Then, we can construct a run R'_j such that from time t_j until time t, p_j is very slow and does not execute any steps, and so, does not send any message—this can be done since processes can be arbitrarily delayed. Process p_k cannot distinguish between R_j and R'_j, and will also send a message to Π_y in R'_j. After t, p_j is back to normal and sends a message to Π_y. Thus, $f_x + f_y + 1$ messages are sent to Π_y, a contradiction.

We now show that $|S| > f_x$. Assume $|S| \leq f_x$. Then, we can construct a run in which $S_x = S$, every p_j in S_x crashes and from our claim above, no other process in $\Pi_x \setminus S$ sends a message to processes in Π_y. Thus, correct processes in Π_x deliver m, and processes in Π_y never receive m, and so, cannot deliver it.

Without loss of generality assume that each process in S_x sends only one message to Π_y. Thus, processes in $S \setminus S_x$ can send up to $f_x + f_y - |S_x| = f_y$ messages to processes in Π_y. Let set S_y denote such processes in Π_y. Since any f_y processes may crash in Π_y, assume that processes in S_y crash in R'. Therefore, in R' correct processes in Π_x deliver m, but correct processes in Π_y do not, a contradiction. □

Even when n_x or n_y are smaller than f_x+f_y+1, our algorithm, which exchanges $(f_x+1)(f_y+1)$ messages is not optimal. Consider the following case in which $n_x = n_y = 4$ and $f_x = f_y = 2$. With our algorithm, processes in Π_x will send 9 messages to processes in Π_y. While this is certainly enough to guarantee correctness, it is possible to do better: instead of sending to three distinct processes in Π_y, each process p_i^x sends a message to processes p_i^y and $p_{(i \,\mathbf{mod}\, 4)+1}^y$. With such an algorithm, only 8 messages are exchanged!

5 The General Case

5.1 A Reliable Broadcast Algorithm without Consistency

Algorithm 3 implements Reliable Broadcast (without the consistency property) for any number of groups and tolerates any number of failures, that is, there is no limit on the number of bad groups and on the number of correct processes in good groups.

Algorithm 3 Reliable broadcast algorithm without the consistency property

Initially:
  $rcvMsgs \leftarrow \emptyset$
  $fwdMsgs \leftarrow \emptyset$

procedure byz-broadcast(m)
  r-broadcast($\{m\}$)

when r-deliver($mset$)
  for all $m \in mset \setminus rcvMsgs$ **do** byz-deliver(m)
  $rcvMsgs \leftarrow rcvMsgs \cup mset$

when $p_i \in \mathsf{senders}_x()$ **and** $rcvMsgs \setminus fwdMsgs \neq \emptyset$
  for all $\Pi_y \in \Pi$ **do** $\mathsf{send}_y([\text{FW}, rcvMsgs \setminus fwdMsgs])$
  $fwdMsgs \leftarrow rcvMsgs$

when $\mathsf{receive}_y([\text{FW}, mset])$
  r-broadcast($mset$)

Algorithm 3 builds on Algorithm 2. It works as follows. In order for some process in a good group to byz-deliver a message, it has to r-deliver it (i.e., using local Reliable Broadcast). This guarantees that all correct processes in the group will r-deliver the message. Using a mechanism similar to the one used in Algorithm 2, the message will eventually reach some correct process in each good group, which will r-broadcast the message locally, and also propagate it to other groups. In principle, the inter-group communication of Algorithm 3 is similar to the Reliable Broadcast algorithm presented in [CT96], for a "flat" process model.

5.2 A Reliable Broadcast Algorithm with Consistency

We now extend Algorithm 3 to also enforce the consistency property. Algorithm 4 resembles Algorithm 3. The main differences are the first *when* clause, the fact that all messages are signed to guarantee authenticity, and the fact that the local Reliable Broadcast requires the consistency property.

Local Reliable Broadcast with the consistency property can be implemented with an algorithm similar, in principle, to Algorithm 4. To r-broadcast some message m, p_i in Π_x signs m and sends it to all processes in Π_x (we use $mset : k_i$ to denote that the message set $mset$ is signed by p_i). When a process p_j receives m for the first time, it also signs m and sends it to all processes in Π_x. If a process receives m from $\lceil (2n+1)/3 \rceil$ processes, it r-delivers m. (A detailed description of this algorithm is given in the Appendix.)

Algorithm 4 Reliable broadcast algorithm with the consistency property

```
Initially:
   rcvMsgs ← ∅
   fwdMsgs ← ∅
   dlvMsgs ← ∅

procedure byz-broadcast(m)
   r-broadcast({m} : k_i)

when r-deliver(mset : k_j)
   rcvMsgs ← rcvMsgs ∪ mset
   for each m ∈ rcvMsgs \ dlvMsgs do
      if [for ⌈(2n + 1)/3⌉ groups Π_y, ∃p_l ∈ Π_y : r-delivered (mset′ : k_l) and m ∈ mset′] then
         byz-deliver(m)
         dlvMsgs ← dlvMsgs ∪ {m}

when p_i ∈ senders_x() and rcvMsgs \ fwdMsgs ≠ ∅
   for each Π_y ∈ Π do send_y([FW, (rcvMsgs \ fwdMsgs) : k_i])
   fwdMsgs ← rcvMsgs

when receive_y([FW, mset : k_j])
   r-broadcast(mset : k_j)
```

References

[BT85] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4), October 1985.

[Car99a] L. Cardelli. Abstractions for mobile computation. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*. Springer Verlag, 1999.

[Car99b] L. Cardelli. Wide area computation. In *Proceedings of the 26th International Colloqium on Automata, Languages, and Programming (ICALP)*, 1999. LNCS 1644.

[CHT96] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[CT90] T. D. Chandra and S. Toueg. Time and message efficient reliable broadcasts. In *Proceedings of the 4th International Workshop on Distributed Algorithms*, September 1990.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[GS96] R. Guerraoui and A. Schiper. Consensus service: A modular approach for building fault-tolerant agreement protocols in distributed systems. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, pages 168–177, Sendai, Japan, June 1996.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, chapter 5. Addison-Wesley, 2nd edition, 1993.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2), April 1980.

[Rei94] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicasts in rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, 1994.

Appendix: Proofs

Algorithm 1: Implementation of $\mathsf{send}_x()$ and $\mathsf{receive}_x()$

Proposition 2 (Termination.) *The* $\mathsf{send}_x()$ *abstraction is non-blocking.*

Proof: Follows from the fact that the $\mathsf{destinations}_x()$ abstraction, the send primitive, and the **fork** operation are all non-blocking. □

Proposition 3 (Validity.) *If a correct process in* Π_y *invokes* $\mathsf{send}_x()$ *with a message* m *then eventually a correct process in* Π_x *can receive* m *by invoking* $\mathsf{receive}_x()$*.*

Proof: Assume that a correct process p_i in Π_y invokes $\mathsf{send}_x()$ with a message m. There are now two cases to consider: (a) the *dest* set in line 3 contains a correct process p_j or (b) the *dest* set does not contain a correct process. With case (a), p_i will send m to p_j. By the properties of the send and receive primitives, p_j can eventually receive m. Consider now case (b). By the Validity property of $\mathsf{destinations}_x()$, the $\mathsf{destinations}_x()$ abstraction eventually returns a correct process p_j in Π_x. Since p_j is not contained in *dest* initially, there is an invocation of $\mathsf{destinations}_x()$ that returns p_j and where the test in line 7 becomes true. In this iteration of the **while** loop, p_i will send [GS,m] to p_j. The Proposition then follows from the properties of the send and receive primitives as above in case (a). □

Proposition 4 (Integrity.) *If* $\mathsf{receive}_x()$ *returns a message* m *to an honest process* p*, and if sender(m) is honest, then sender(m) called* $\mathsf{send}_x()$ *with* m*.*

Proof: Given an honest process p that calls $\mathsf{receive}_x()$ and thereby receives a message m. Assume that sender(m) is honest. For $\mathsf{receive}_x()$ to return m at p, p must have received a message of the form [GS,m]. Since sender(m) is honest, sender(m) only sends such a message as part of the $\mathsf{send}_x()$ abstraction. □

Algorithm 2: Reliable Broadcast for Two good Groups

Proposition 5 (Validity.) *If a correct process* p_i *in a good group* Π_x *byz-broadcasts a message* m*, then it byz-delivers* m*.*

Proof: Since p_i byz-broadcasts m, it r-broadcasts m. From validity of reliable broadcast, it eventually r-delivers some message $mset$ such that $m \in mset$. Assume for the sake of a contradiction that p_i does not byz-deliver m. So, it must be that p_i has included m in $rcvMsgs$, but when this happens, p_i byz-delivers m, a contradiction that concludes the proof. □

Proposition 6 (Agreement.) *If a correct process* p_i *in a good group* Π_x *byz-delivers* m*, then for every good group* Π_y *and each correct process* p_j *in* Π_y*,* p_j *also byz-delivers* m*.*

Proof: Assume for a contradiction that p_j does not byz-deliver m. There are two cases to consider: (a) p_j and p_i are in the same group or (b) p_j and p_i are in different groups.

- Case (a). Since p_i has byz-delivered m, by lines 6–7, p_i has r-delivered m. From agreement of reliable broadcast, eventually p_j also r-delivers m. Since p_j does not byz-deliver m, m must already be part of the set $rcvMsgs$. Consider the execution of the **when** clause in line 6 when m is added to $rcvMsgs$. In this execution, p_j byz-delivers m, which is a contradiction.

- Case (b). There is a time t after which the $\mathsf{senders}_x()$ abstraction returns the same set of correct processes for every invocation. Consider a correct process p_r in Π_x that is part of this set. There are two subcases to consider: (b.1) p_r invokes the $\mathsf{send}_x()$ abstraction with m in line 10 and (b.2) p_r does not invoke the $\mathsf{send}_x()$ abstraction with m in line 10.

  Consider first case (b.1). The properties of $\mathsf{send}_x()$ and $\mathsf{receive}_x()$ ensure that a correct process p_k in Π_y can receive m with $\mathsf{receive}_x()$. When p_k receives m, p_k sends m to all processes in Π_y including p_j. Because both p_k and p_j are correct, the properties of send and receive ensure that p_j will receive m. When p_j receives m, p_j also byz-delivers m, which is a contradiction.

  Consider next case (b.2). The set $rcvMsgs_r \setminus fwdMsgs_r$ never contains m after t. Because of the agreement of Local Reliable Broadcast, p_r will eventually r-deliver m and add it to $rcvMsgs_r$. Thus, the set $fwdMsgs_r$ contains m before t. However, we only add to the set $fwdMsgs_r$ in line 11, and when m is added to the set in line 11, p_r invoked $\mathsf{send}_x()$ with m in line 10, which is a contradiction.

□

Proposition 7 (Integrity.) *Each honest process p_i byz-delivers every message at most once. Moreover, if sender(m) is honest, then sender(m) byz-broadcast m.*

PROOF: There are two cases to consider: (a) p_i and sender(m) are in the same group and (b) p_i and sender(m) are in different groups.

- Case (a). If p_i is in the same group as sender(m), then p_i byz-delivers m in line 7. Thus, p_i also r-delivers m. Integrity of Local Reliable Broadcast implies that sender(m) r-broadcasts m. According to the algorithm, this only happens if sender(m) byz-broadcasts m. The at-most-once byz-delivery by p_i follows from the assignment in line 8: if p_i byz-delivers m, then the assignment adds m to $rcvMsgs$.

- Case (b). If p_i is in a different group than sender(m), then p_i byz-delivers m in line 15. Thus, p_i also receives [LC,m]. Integrity of send and receive guarantees that some process p_r sent [LC,m] in line 13. Moreover, the integrity of $\mathsf{send}_x()$ and $\mathsf{receive}_x()$ ensures that some process p_k in Π_x invoked $\mathsf{send}_x()$ with m. This means that p_k added m to its $rcvMsgs$ set, which only happens if p_k r-delivers m. The integrity of Local Reliable Broadcast now ensures that some process p_m in Π_x invoked byz-broadcast with m. The at-most-once byz-delivery follows from the integrity of the various primitives and abstractions and from the assignment in line 11, which ensures that a process never invokes $\mathsf{send}_x()$ twice with the same message.

□

Algorithm 3: Reliable Broadcast Without Consistency

Proposition 8 (Validity.) *If a correct process p_i in a good group Π_x byz-broadcasts a message m, then it byz-delivers m.*

PROOF: Similar to the proof of Proposition 5. □

Proposition 9 (Agreement.) *If a correct process p_i in a good group Π_x byz-delivers m, then for every good group Π_y and each correct process p_j in Π_y, p_j also byz-delivers m.*

PROOF: Assume for a contradiction that p_j does not byz-deliver m. We claim that p_j does not r-deliver any set $mset$ containing m. Denote such a set $mset(m)$. If p_j r-delivers $mset(m)$, and does not byz-deliver m, then from lines 6–8, it has to be that $m \in rcvMsgs$. But $rcvMsgs$ is initially empty, and so, p_j included m in $rcvMsgs$. This can only happen at line 8 if $m \in mset$; right before this happens, p_j executed line 7 such that $m \notin rcvMsgs$. Thus, p_j byz-delivers m at this time. Therefore, p_j does not r-deliver any set $mset(m)$. There are two cases: (a) p_j and p_i are in the same group, (b) p_j and p_i are in different groups.

- Case (a). Since p_i has byz-delivered m, by lines 7–8, p_i has r-delivered m. From agreement of reliable broadcast, eventually p_j also r-delivers m, a contradiction.
- Case (b). Since p_j is correct, from the reliable broadcast properties, no correct process p_k in Π_y r-broadcasts a message $mset(m)$—otherwise p_j would r-deliver $mset(m)$, and so, no such process executes $\mathsf{receive}_y$([FW, $\{mset(m)\}$]).

  Let p_r be some correct process in Π_x. Thus, from the $\mathsf{send}_x()$ and $\mathsf{receive}_x()$ abstractions, p_r does not execute send_y([FW, $mset(m)$]). From case (a), $m \in rcvMsgs_r$, and so, it must be that p_r does not execute send_y([FW, $mset(m)$]).

  Variable $fwdMsgs_r$ is initially empty, and is only updated by p_r with some message after p_r sends this messages to other groups (lines 10–11). Therefore, eventually $m \in rcvMsgs_r \setminus fwdMsgs_r$ and from line 9, p_r is never in $\mathsf{senders}_x()$. It follows that no correct process is ever selected by $\mathsf{senders}_x()$, a contradiction. □

Proposition 10 (Integrity.) *Each honest process p_i byz-delivers every message at most once. Moreover, if sender(m) is honest, then sender(m) byz-broadcast m.*

PROOF: Messages are all byz-delivered at line 7 and only if they are not in $rcvMsgs$. Right after byz-delivering a message, unless it fails, every honest process includes in $rcvMsgs$. Thus, no message is byz-delivered more than once. From the algorithm, it follows immediately that if sender(m) is honest, then sender(m) byz-broadcast m. □

Algorithm 4: Reliable Broadcast With Consistency

Proposition 11 (Validity.) *If a correct process p_i in a good group Π_x byz-broadcasts a message m, then it byz-delivers m.*

Proof: To byz-broadcast m, p_i signs it and r-broadcasts the signed message (lines 5–6). From validity of reliable broadcast, p_i eventually r-delivers some message $mset$ such that $m \in mset$—we denote such a set $mset(m)$. From agreement of reliable broadcast, every correct process in Π_x r-delivers some set $mset(m)$. From lines 13–15 and the fact that $\mathsf{senders}_x()$ eventually outputs some correct process in Π_x, some correct process in Π_x will execute $\mathsf{send}_y([\text{FW}, mset(m)])$, for every group Π_y in Π. From the properties of the $\mathsf{send}_x()$ and $\mathsf{receive}_x()$ abstractions, some correct process in each good group will receive $mset(m)$ (line 16), and locally r-broadcast it. Applying a similar argument, it follows that eventually every good group executes $\mathsf{send}_y([\text{FW}, mset(m)])$, for every group Π_y in Π. Since there are $\lceil (2n+1)/3 \rceil$ good groups, each correct in each good group will r-deliver $\lceil (2n+1)/3 \rceil$ sets of the type $mset(m)$, signed by some process in a good group. Thus, p_i byz-delivers m. □

Proposition 12 (Agreement.) *If a correct process p_i in a good group Π_x byz-delivers m, then for every good group Π_y and each correct process p_j in Π_y, p_j also byz-delivers m.*

Proof: If p_i byz-delivers m, then it has r-delivered $\lceil (2n+1)/3 \rceil$ sets $mset$ containing m, and signed by processes from different groups. Thus, p_i r-delivered such a set signed by at least one process in a good group Π_z. It follows that this process, or some other process in Π_z sends some set with m to all groups. Therefore, it can be shown that every correct process in each good group receives $\lceil (2n+1)/3 \rceil$ sets $mset$ containing m. From lines 7–12, such a process byz-delivers m. □

Proposition 13 (Integrity.) *Each honest process p_i byz-delivers every message at most once. Moreover, if $sender(m)$ is honest, then $sender(m)$ byz-broadcast m.*

Proof: Similar to the proof of Proposition 10. □

Proposition 14 (Consistency.) *Let p_i and p_j be two processes in good groups. If p_i byz-delivers m, p_j byz-delivers m', and $id(m) = id(m')$, then $val(m) = val(m')$.*

Proof: For a contradiction, assume that p_i and p_j deliver messages m and m', respectively, and even though $id(m) = id(m')$, $val(m) \neq val(m')$. From lines 11–12, both p_i and p_j r-delivered $\lceil (2n+1)/3 \rceil$ sets $mset$ containing m and signed by processes in different groups. Since there are at most $\lfloor (n-1)/3 \rfloor$ malicious processes, there must be at least one good group Π_z from which both p_i and p_j r-delivered a signed message with $mset(m)$ and $mset(m')$, respectively. Let p_r and p_s be processes in Π_z that signed, respectively, the messages $mset(m)$ and $mset(m')$. Before p_r signs and sends $mset(m)$ to some process in p_i's group, it r-delivered $mset(m)$. Likewise, before p_s signs and sends $mset(m')$ to some process in p_j's group, it r-delivered $mset(m)$. Since $id(m) = id(m')$ and $val(m) \neq val(m')$, consistency of local reliable broadcast is violated—a contradiction that concludes the proof. □

Algorithm 5: Solving Local Reliable Broadcast With Consistency

Algorithm 5 solves local Reliable Broadcast. We omit the correctness proof since it is similar to the proof of Algorithm 4.

Algorithm 5 Local Reliable Broadcast with consistency

Initially:
  $rcvMsgs \leftarrow \emptyset$
  $dlvMsgs \leftarrow \emptyset$

procedure r-broadcast(m)
  send $m : k_i$ to all

when receive $m : k_j$
  if $m \notin rcvMsgs$ **then**
    send $m : k_i$ to all
    $rcvMsgs \leftarrow rcvMsgs \cup \{m\}$
  for each $m \in rcvMsgs \setminus dlvMsgs$ **do**
    if [for $\lceil (2n+1)/3 \rceil$ processes p_r: received $m : k_r$ from p_r] **then**
      r-deliver(m)
      $dlvMsgs \leftarrow dlvMsgs \cup \{m\}$