OldBlue: Causal Broadcast In A Mutually Suspicious Environment (Working Draft)

Matthew D. Van Gundy
Department of Computer Science
University of California, Davis, CA, USA
mdvangundy@ucdavis.edu

Hao Chen
Department of Computer Science
University of California, Davis, CA, USA
hchen@cs.ucdavis.edu

ABSTRACT

Many protocols have been proposed to provide reliability and consistency guarantees for group-oriented communication in distributed systems. However, existing systems tolerate only benign failures or a limited number of Byzantine failures. This limitation is problematic for systems consisting of mutually suspicious Internet peers not controlled by a centralized authority.

We propose OldBlue, a broadcast protocol for distributed systems that: ensures causal delivery ordering over all messages, guarantees that any two correct processes will have consistent views of the full causal history of any message delivered to both processes, remains secure in the presence of an arbitrary number of Byzantine failures, and allows connected correct processes to make progress during periods of network partition. To our knowledge, OldBlue is the first protocol which provides these guarantees in such a strong threat model. We provide proofs that OldBlue meets its formal requirements and present simulation results indicating an upper-bound on its expected performance in practice.

General Terms

Security, Algorithms, Reliability

Keywords

Causal Broadcast, Distributed Systems, Consensus, Byzantine Adversary

1. INTRODUCTION

With the rising popularity of the peer-to-peer (P2P) networking paradigm (e.g. Chord, Kademlia, Skype, BitTorrent), an increasing number of large distributed systems are being constructed from computing and network resources contributed by volunteers throughout the globe. P2P's widely distributed nature and lack of central control make it an attractive architecture for building anonymity-preserving and censorship-resistant networks such as Tor and Freenet. However, adversaries can volunteer resources to P2P systems in order to attack from within. Because there is no a priori bound on the level of adversarial influence, participants in P2P networks find themselves in a setting of "mutual suspicion" — every other participant may be attempting to thwart the goals that the system seeks to achieve. This has led to some efforts to design secure P2P network variants [22, 15, 14]. However, many systems could benefit if fundamental distributed systems primitives could be made secure in this setting.

One of the most fundamental primitives for group oriented communication is Causal Broadcast. A Causal Broadcast protocol is a broadcast network protocol which ensures that messages are delivered in an order that preserves the potential causal relationships between messages. Thus, if message $\mathtt{m}_1$ could have caused $\mathtt{m}_2$ (the author of $\mathtt{m}_2$ sent or delivered $\mathtt{m}_1$ before sending $\mathtt{m}_2$), a process will not deliver $\mathtt{m}_2$ unless that process has already delivered $\mathtt{m}_1$. Typically, this property is enforced by including in each message either a vector timestamp or the unique message identifiers of causally preceding messages [16, 18]. This allows the recipient of a message $\mathtt{m}_2$ to precisely determine the set of messages M which causally precede $\mathtt{m}_2$ and to ensure that $\mathtt{m}_2$ will only be delivered if all messages in M have been delivered.

Many Causal Broadcast protocols exist. However, most tolerate only benign failures. [13, 1, 16, 9, 4, 17] Solutions based on tamper-proof hardware which tolerate Byzantine failures have been presented. [20, 21] However, such hardware is still not in wide deployment on common commodity PCs. Causal Broadcast can be achieved by building on protocols providing stronger guarantees, such as Reliable Broadcast [10, 7, 19, 2, 9] and Byzantine Agreement [11, 12, 8]. However, this is undesirable. Though, Byzantine Agreement is solvable for $t = n - 1$ Byzantine failures in general, practical Byzantine Agreement and Reliable Broadcast protocols that guarantee liveness among correct processes, can tolerate at most $t \leq \lfloor \frac{n-1}{3} \rfloor$ Byzantine failures. [6] P2P network churn and temporary network outages can cause large numbers of processes to be temporarily unreachable. During a temporary outage of $t \geq \lfloor \frac{n-1}{3} \rfloor$, Reliable Broadcast protocols will be unable to send and deliver new messages and Byzantine Agreement protocols must deliver failure messages while a Causal Broadcast protocol can allow processes within each connected component to continue sending and delivering messages to one another because each component can ensure causal ordering locally. In addition to temporary network outages, the $t \leq \lfloor \frac{n-1}{3} \rfloor$ limit is inconsistent with the mutually suspicious setting of P2P networks where an attacker can control an arbitrary fraction of system nodes limited only by the attacker's resource constraints. More general Byzantine Agreement protocols fare no better. Laying efficiency considerations aside, if we allow up to $n - 1$

(v. e24eaa3bd090+)

Byzantine failures, a Byzantine Agreement protocol must deliver a failure message if even a single process goes offline. This indicates the need for a protocol that directly ensures Causal Broadcast, is tolerant of an arbitrary number of Byzantine failures, and ensures liveness among connected processes during a network partition.

OldBlue ensures causal ordering by identifying each message by a message digest over the sender, the identifiers of each of the message's causal predecessors, and the message payload. Because each message embeds the identifiers of all immediate causal predecessors, recipients of a message `m` can ensure that they have delivered all messages causally preceding `m` before delivering `m`. Origin authenticity is guaranteed through the use of digital signatures and, because message identifiers form a Merkle hash tree over all preceding messages, the recipient of `m` is guaranteed that, once they deliver `m`, their view of `m` and all of `m`'s causal predecessors is consistent with the sender's view. Processes in OldBlue cooperate in order to opportunistically retransmit lost messages.

Like Reiter and Gong's Piggybacking protocol [18], OldBlue piggybacks message digests to ensure that recipients can precisely determine causal predecessors of each message. Like Psync, OldBlue uses the directed acyclic graph structure of the causal precedence relation to reduce the amount of information that must be sent in each message. Unlike either protocol, OldBlue also enforces a formal fairness property and goes to great lengths to prevent starvation between correct connected processes despite allowing an arbitrary number of Byzantine processes. In contrast to protocols which give stronger delivery and ordering guarantees, such as Byzantine Agreement, OldBlue's direct support of causal precedence and fairness allows connected correct processes to make progress during times of network partition.

2. THREAT MODEL

We first consider the setting in which OldBlue operates. A protocol session takes place between n distinct processes. For simplicity, we assume the processes are connected by an unreliable asynchronous broadcast medium, which may be simulated via point-to-point messages or multicast.

The degree to which the network is unreliable is a function of a Byzantine adversary that has complete control over the network. The adversary may modify, insert, delete, duplicate, and reorder messages as she pleases. This ensures that OldBlue's security properties are not dependent on the properties of the underlying network. In any network structured as a spanning tree, for any two processes, there will be a single device (i.e. server, router) mediating all traffic between them. On Ethernet local area networks, an adversary may be able to use ARP spoofing to position himself to mediate all traffic on the local broadcast domain.

The adversary may also *corrupt* an arbitrary number (t) of processes, learning their internal state including both their long-lived and session secrets. The adversary can cause corrupt processes to deviate from the protocol arbitrarily. In contrast to corrupt processes, *correct* processes neither divulge their internal state nor deviate from the protocol. Processes with incorrect implementations or hardware failure are considered corrupt and their behavior to be the manner in which the adversary has caused them to deviate from the protocol. Also note that, because the adversary has full control of the network, she may prevent a *correct* process p from receiving any message(s) sent by other session processes, effectively disconnecting them from the network.

In the mutually suspicious setting of many P2P networks, an adversary may join as many nodes to the system as her resources allow. While place no limits on the number of corrupt processes (t), some of OldBlue's properties are trivially true unless there are at least two correct processes. Therefore, in the rest of the paper we will assume $t \leq n - 2$.

3. PROTOCOL PROPERTIES

OldBlue provides the following interface to other system layers:

`CB.open(processes, pid, gk)` begin a causal broadcast session among `processes` (initialize state, etc.). `pid` is the identifier of the local process. `gk` is the session group encryption key.

`id CB.broadcast(msg)` invoked by the application layer to instruct to broadcast message `msg` to all processes. Returns a unique `id` which distinguishes the current invocation at the local process from all other invocations by all other processes. This permits associating each invocation of `CB.deliver()` below with a specific invocation of `CB.broadcast()`. Because an application may send messages in response to inputs provided by the adversary, the adversary is free to invoke the `CB.broadcast(m)` method of correct processes for arbitrary `m`.

`CB.deliver(sender, id, msg)` callback to application layer signaling delivery of message `msg` with id `id` authored by `sender`. Messages are delivered in causal order.

`CB.close()` shutdown of a session opened by `CB.open()`.

We define the formal properties of OldBlue in terms of the operation of `CB.broadcast()` and `CB.deliver()`. As a Causal Broadcast protocol, we must first review the notion of causal order before discussing other protocol properties. Informally, if receipt of a message `m` could have caused the sending of message `m′`, we say that `m` causally precedes `m′`. A formal definition follows:

Definition 1. Causal Precedence. An event *a causally precedes* an event b (written $a \rightarrow b$) if and only if one of the following conditions hold:

1. a correct process p executes `i = CB.broadcast(m)` (event a) and b is the corresponding message delivery `CB.deliver(p, i, m)` (i.e. broadcast of a message precedes its delivery)
2. a correct process executes `CB.broadcast(m)` (event a) before executing `CB.broadcast(m')` (event b) (i.e. temporal progression at a correct process)
3. a correct process executes `CB.deliver(p, i, m)` (event a) then `CB.broadcast(m')` (event b) (i.e. delivery of `m` precedes broadcast of `m′`)
4. there exists some event e such that $a \rightarrow e$ and $e \rightarrow b$ (i.e. transitive closure)

The causal precedence relation $\cdot \rightarrow \cdot$ defines a partial ordering over events. If $a \rightarrow b$, then we can say that a happened before b. Any two events c and d not related by the

causal precedence relation are said to be *concurrent* (i.e. c and d are concurrent if $c \not\to d$ and $d \not\to c$). No statements can be made about the execution order of concurrent events.

In the context of Causal Broadcast protocols, all events of interest correspond to broadcast or delivery of messages. Therefore, we will abuse notation and speak of a message `m` causally preceding another message $\mathtt{m}'$ ($\mathtt{m} \to \mathtt{m}'$) where event a is the broadcast or delivery of `m`, event b is the broadcast of $\mathtt{m}'$ and $a \to b$ according to Definition 1.

OldBlue captures potential causality between messages. It may be the case that a message `m` was delivered before a message $\mathtt{m}'$ was broadcast, but that the broadcast of $\mathtt{m}'$ was in no way influenced by `m`. It is not possible to distinguish this scenario without application-specific knowledge. Therefore, if a message `m` *could* have influenced the sending of another message $\mathtt{m}'$, OldBlue conservatively determines that $\mathtt{m} \to \mathtt{m}'$.

We restrict the definition of causal precedence to the actions performed by correct processes. Incorporating the internal actions of corrupt processes into the definition of causal precedence is not meaningful because corrupt processes can behave arbitrarily. We impose a distinction between *actual* causal precedence — as defined above by the behavior of correct processes — and *apparent* causal precedence. Causal precedence between messages must be represented in protocol messages. It is possible that corrupt processes may author messages which *appear* to have an arbitrary set of (possibly non-existent) causal predecessors. Therefore, message `m` with id `i` authored by corrupt process p enters the causal precedence relation only when a correct process executes `CB.deliver(p, i, m)`. We address this issue further in Section 4.

The formal guarantees that OldBlue provides are captured in the following definitions:

Definition 2. Validity. If a correct process p executes `i = CB.broadcast(m)` for a message `m`, then p eventually executes `CB.deliver(p, i, m)`. If the adversary delivers all messages associated with `CB.broadcast(m)` to a correct process q, q will execute `CB.deliver(p, i, m)`.

Definition 3. Causal Consistency. No correct process p will execute `CB.deliver(q, i, m)`, until it has delivered all messages $\mathtt{m}'$ which causally precede `m`. Specifically, if q (the author of `m`) is a correct process, before executing `CB.deliver(q, i, m)`, p will have executed:

1. `CB.deliver(x, i', m')` for each such call that occurred at q before q executed `i = CB.broadcast(m)`
2. `CB.deliver(q, i', m')` for each call `i' = CB.broadcast(m')` made by q before q executed `i = CB.broadcast(m)`

Definition 4. Authenticity. Every correct process executes `CB.deliver(p, i, m)` at most once for each value of `i` (and any values of p and `m`) and, if p is correct, then p previously executed `i = CB.broadcast(m)`.

Validity provides three guarantees. It rules out trivial protocols that deliver no messages. It ensures self-delivery of messages. And, it ensures liveness of connected correct processes—if the adversary delivers all protocol messages associated with an invocation of `CB.broadcast(m)` to a correct process, that process must deliver `m`. A message is associated with `CB.broadcast(m)` if it is the initial broadcast of `m`, a request to retransmit a lost message causally preceding `m` sent by a process trying to deliver `m`, or a message causally preceding `m` retransmitted in response to such a retransmission request. If the adversary does not faithfully deliver all such associated messages, to a correct process q, q is not required to deliver `m`. (Indeed, in some cases to do so would violate Causal Consistency.)

Causal Consistency ensures that all messages are delivered in causal order. Causal Consistency further ensures that, whenever any two correct processes execute `CB.deliver(q, i, m)`, these two processes agree on `m` and all $\mathtt{m}' \to \mathtt{m}$ — over both causal ordering and message contents — without regard to the correctness of the authors of `m` or its causal predecessors. This is particularly useful, because it allows us to know something about the level of consistency between the states of two processes even if communication is one-way or no messages ever become stable.

If q, the author of `m`, is corrupt, the causal precedence of `m` will not be defined until it `m` has been delivered by a correct process. However, `m` will have an apparent set of causal predecessors. Any correct process p that delivers `m` does not know whether `m`'s author is correct. Therefore, p must deliver `m` subject to the ordering imposed by `m`'s apparent causal predecessors to ensure that p's operation is correct irrespective of the correctness of q. A corrupt author q does not weaken the consistency guarantees ensured by Definition 3. To see this, let p execute `CB.deliver(q, i, m)`. Let the next message that p sends be $\mathtt{m}'$. `m` is now a causal predecessor of $\mathtt{m}'$. Therefore, Validity requires any correct process r which receives the messages associated with `i' = CB.broadcast(m')` to execute `CB.deliver(p, i', m')`. Because p is correct, Causal Consistency also requires r to `CB.deliver(q, i, m)` and all apparent causal predecessors delivered by p. Thus Causal Consistency guarantees that correct processes p and r will agree on $\mathtt{m}'$ and all causal predecessors (including `m`) after the delivery of $\mathtt{m}'$ regardless of the correctness of q, the author of `m`.

In this way, Causal Consistency extends the liveness property ensured by Validity. Validity ensures that processes will be able to deliver a faithfully transmitted message from a correct process. Causal Consistency ensures that all previously undelivered causally preceding messages will also be delivered at that time.

Authenticity ensures that any attempt to impersonate a correct process or replay messages will not succeed. Authenticity also ensures that, for every delivered message, if the purported author is correct, the message was transmitted without modification from that author. Because corrupt processes can deviate arbitrarily from the protocol, Authenticity does not place any constraints on their internal behavior.

Corrupt processes may attempt to monopolize the resources of correct processes to prevent them from making progress. To prevent this from occurring, each process explicitly or implicitly requests the information they needed in order to make progress. A request is either the retransmission of a message explicitly requested by another process, the transmission of a new message from the local process (all processes that do not make an explicit request for a retransmission are considered to have implicitly requested new messages), or an outgoing request from the local process for other processes to retransmit a message. Correct processes then impose a fair scheduling criterion over their outgoing

messages in order to fulfill the requests of other processes in turn.

We associate with every process p a FIFO request queue R_p : *Requests*. We define a function `bool eligible(process, request)` that captures external constraints governing whether or not a process is able to accept the fulfillment of `request` at the time of invocation. E.g. A congestion control mechanism might dictate that process p is congested, thus we should not attempt to send a message requested by p right now because it is likely to be dropped. We assume that, for each p and request r, `eligible(p, r)` depends only on information controlled by p and network conditions. Thus, if p is correct, the adversary cannot cause p to be ineligible by means other than exercising her ability to drop and delay messages on the network. Furthermore, we stipulate that eventually `eligible(p, r)` will become true for all processes p and requests r.

Definition 5. Fairness. A scheduling algorithm outputs a sequence of requests from processes $(p_1, r_1), (p_2, r_2), \ldots$ We call a scheduling algorithm fair if each process with eligible requests will have one of their requests scheduled at least once in every n requests. That is to say, for each process p where p has an eligible request at the time the i^{th} request (r_i) is scheduled, there is a request r_j with $p_j = p$, $r_j \in R_p$, and $i \leq j < i + n$.

Fairness ensures that corrupt processes cannot cause a correct process to starve other correct processes. We restrict fairness to the outgoing messages sent by each correct process because processes cannot control which messages they receive, nor can they determine the author of a message until after signatures are verified. Correct processes should process incoming messages in a first-come first-served order. We assume that differences in computation time and message transmission time due to message length can be upper bounded by some suitably small constant allowing message-based fairness to be approximately equivalent to definitions that take computation or bandwidth into account.

4. THE OldBlue PROTOCOL

In this section we describe the OldBlue protocol. OldBlue is positioned above a network transport protocol layer that supports unreliable asynchronous unordered end-to-end message transmission via multicast (possibly simulated using multiple unicasts). `send(recipients, msg)` will multicast `msg` to recipients. The callback `recv(sender, msg)` is invoked when a message, allegedly from `sender`, is received from the network.

We associate with each process p a public-private signing key pair (`pk(`p`)`, `sk(`p`)`). We assume that the correspondence between public signing keys and process identities is known to all processes. We denote signing (respectively verifying) message `m` under key `sk(`p`)` (`pk(`p`)`) by σ `= S(sk(`p`), m)` (`V(pk(`p`), m, `σ`)`). We denote encryption and decryption of message `m` under key `gk` by `E(gk, m)` and `D(gk, m)` respectively.

We assume the existence of unambiguous serialization and deserialization functions `encode()` and `decode()` with `m = decode(encode(m))` which, respectively, are able to encode a data type for transmission over the network and decode a corresponding language-level data type from the network. We assume that `H()` is a collision-resistant cryptographic hash function such that it is computationally infeasible to find two inputs `m1 != m2` such that `H(m1) == H(m2)`.

Each process maintains the following state for each ongoing OldBlue session:

```
class CB:
  Processes   : Set of ProcId
  Pid         : ProcId
  Requests    : Queue of (ProcId, Set of Request)
  Delivered   : Map from MsgId to Message
  Undelivered : Map from MsgId to Message
  Frontier    : Set of MsgId
  Wire        : Map from MsgId to String
  gk          : Encryption Key
```

`Processes` identifies the processes that are members of the OldBlue session. `Pid` is the identity of the local process. `Requests` is a priority queue of process id's and the outstanding requests associated with that process maintained in least-recently-used order. `Delivered` and `Undelivered` store delivered messages and received but not yet delivered messages for this session respectively. For instance, if a message is received before one of its causal predecessors, it will remain in `Undelivered`. `Frontier` contains the message id's of leaves of the causal graph. If `Frontier` contains the id of message `m`, the local process has not delivered any messages causally newer than `m`. `Wire` stores the on-the-wire representation of network messages indexed by their message id to allow collaborative retransmission. `gk` is the encryption key for the current session shared by session members.

4.1 Initialization

The session begins with a call to `CB.open()` which will initialize all process state for an OldBlue session. The application is responsible for determining session membership and negotiating a shared encryption key for the session. We assume that all correct processes in the session execute `CB.open(processes, pid, key)` with the same value of `processes` and `gk` and differing values of `pid`. The application can ensure this by executing an appropriate Authenticated Group Key Agreement protocol (e.g. [5]) before calling `CB.open()`. `CB.open()` is defined as follows:

```
CB.open(processes, pid, key):
  Processes = processes
  Pid = pid
  Requests = Processes × ∅
  Delivered = ∅
  Undelivered = ∅
  Frontier = ∅
  Wire = ∅
  gk = key
```

4.2 Request Fulfillment

At the highest level, OldBlue works as a request fulfillment engine, fulfilling requests in a fair order as determined by the scheduler. OldBlue's public methods add requests to `Requests` for later fulfillment. Scheduling constraints are encapsulated within the method `schedule()`, which returns an eligible request from `Requests` subject to OldBlue's Fairness constrains. `schedule()` will block the main thread of execution until a request becomes eligible as determined by the `eligible()` function defined in Section 3.

```
class Request:
  mid : MsgId

class Outgoing extends Request
class LostMsg extends Request
class Retransmit extends Request

CB.main():
  repeat forever:
    (owner, request) = schedule(Requests)
    request.fulfill(this, owner)
```

OldBlue defines three kinds of `Requests`. Each `Request` type stores the id of the message it corresponds to and defines a `fulfill()` method which takes the actions necessary to fulfill the request. `Outgoing` requests represent messages `CB.broadcast()` by the local process. `LostMsg` requests indicate causal predecessors required to deliver a message in `Undelivered` which have not been received by the local process. `Retransmit` requests track requests by other processes to retransmit lost messages.

4.3 Message Transmission

Transmission of new messages (Figure 1) in OldBlue is straightforward. Each `Message` encodes the author, the message ids of all immediately causally preceding messages, and the message payload provided by the application. The message id of each message uniquely identifies the message within the session by taking a digest over the author, the ids of all causal predecessors, and the message payload. Immediately self-delivering the message ensures that the self-delivery requirement of Validity is satisfied. `CB.broadcast()` adds an `Outgoing` request for the message id to the request queue of the local process.

Fulfillment of the `Outgoing` request actually broadcasts the message to the other processes in the session. A standard Encrypt-then-Sign construction [3] is used to prevent forgery of messages by corrupt processes and to preserve confidentiality and authenticity despite external adversaries. To facilitate collaborative retransmission, the author's identity is included in the network-level payload which is also saved in `Wire` for future retransmissions.

`deliverMessage(m)` performs the actual delivery of deliverable message `m`. `m` is added to `Delivered`. `m` replaces any of its immediate causal predecessors in `Frontier` and, finally, `m` is provided to the application via `CB.deliver()`.

4.4 Message Receipt

When a network-level message is received (Figure 2), its signature is verified and it is decrypted. If any error occurs in deserialization, signature verification, or decryption, `authDecrypt()` returns null. Otherwise, it returns a reconstituted `Message` or `LostMsg` object (we deal with `LostMsg` in Section 4.5).

`Message.receive()` handles the actual processing of the message. It first performs sanity checks to reject messages with incorrect author or id fields, senders outside of the process group, or previously delivered messages. It then removes any requests to retransmit the current message that the local process may have initiated. If the new message had any immediate causal predecessors that have not been received, a `LostMsg` request is added to the request queue for the local process. The arrival of a message may cause messages in `Undelivered` to become deliverable. Therefore, the

```
class Message:
  author     : ProcId
  parentids  : Set of MsgId
  payload    : String
  id         : MsgId

id CB.broadcast(msg):
  m = Message(author=Pid, payload=msg)
  m.parentids = Frontier // causal preds
  m.id = mid(m)
  deliverMessage(m)      // Self-delivery
  addRequest(Pid, Outgoing(mid=m.id))
  return m.id

MsgId mid(msg):
  return H(msg.author, msg.parentids,
           msg.payload)

deliverMessage(m):
  Delivered[m.id] = m
  for parent ∈ m.parentids:
    Frontier = Frontier \ { parent }
  // m is now a leaf of the causal graph
  Frontier = Frontier ∪ { m.id }
  CB.deliver(m.author, m.id, m.payload)

Outgoing.fulfill(cb, owner):
  msg = cb.Delivered[mid]
  cb.Wire[mid] = cb.authEncrypt(msg)
  send(cb.Processes, cb.Wire[mid])

String authEncrypt(sender, payload):
  ctxt = E(gk, encode(payload))
  sig = S(sk(Pid), ctxt)
  return encode(Pid, ctxt, sig)

// Add req to request queue for process p
addRequest(p, req):
  Find i s.t. Requests[i] == (p, R)
  Requests[i] = (p, R ∪ { req })

// Remove any requests by p equal to req
removeRequest(p, req):
  Find i s.t. Requests[i] == (p, R)
  Requests[i] = (p, R \ { req })
```

Figure 1: Implementation of CB.broadcast()

new message is added to the set of `Undelivered` messages and all deliverable undelivered messages are delivered.

4.5 Message Loss and Retransmission

When a `Message` arrives before one of its causal predecessors, correct processes assume that the predecessor has been lost and they will add a `LostMsg` request to their request queue (Figure 3). Fulfillment of the `LostMsg` request will cause the `LostMsg` request to be forwarded to other processes, serving as a negative acknowledgment and requesting that they retransmit the lost message. The "missing" causal predecessor may not actually be lost, it may merely arrive after a causally newer message making the `LostMsg` request superfluous. We assume that `eligible()` delays `LostMsg` re-

```
Upon recv(sender, message):
  (author, msgOrReq) = cb.authDecrypt(message)
  if msgOrReq != null:
    msgOrReq.receive(cb, author, message)

Object authDecrypt(payload):
  (sender, ctxt, sig) = decode(payload)
  if !V(pk(sender), ctxt, sig):
    return null
  obj = decode(D(gk, ctxt)
  return obj ? (sender, obj) : null

Message.receive(cb, sender, payload):
  // Reject pathologies
  if sender != this.author:  return
  if mid(this) != this.id:  return
  if sender ∉ Processes:  return
  if cb.Delivered[this.id] != null: return

  // Remove requests to retransmit message
  cb.removeRequest(cb.Pid, LostMsg(mid=id))

  // Request missing parents
  for parent ∈ parentids:
    if cb.Delivered[parent] == null
       and cb.Undelivered[parent] == null:
      cb.addRequest(cb.Pid, LostMsg(mid=parent))

  // Add to undelivered set
  cb.Undelivered[id] = this
  cb.Wire[id] = payload
  do:
    anyDelivered = false
    for msg ∈ cb.Undelivered:
      if cb.parentsDelivered(msg):
        // Message is deliverable
        delete cb.Undelivered[msg.id]
        deliverMessage(msg)
        anyDelivered = true
        break // reconsider other undelivered msgs
  while anyDelivered == true

bool parentsDelivered(msg):
  for parentid ∈ msg.parentids:
    if Delivered[parentid] == null:
      return false
  return true
```

Figure 2: Implementation of Message Receipt

quests in an attempt to minimize such superfluous requests.

Because the `LostMsg` request itself may be lost, correct processes re-enqueue the `LostMsg` request, ensuring that the process will continue to request lost messages until they are received. `Message.receive()` will remove the associated `LostMsg` request from the local process's request queue, when the lost message is received.

When a `LostMsg` request is received from another process, a corresponding `Retransmit` request is added to their request queue. When a correct process fulfills a `Retransmit` request, it will forward a copy of the lost message to the requesting process.

```
LostMsg.fulfill(cb, owner):
  payload = cb.authEncncrypt(owner, this)
  send(cb.Processes, payload)
  cb.addRequest(cb.Pid, this)

LostMsg.receive(cb, sender, payload):
  if sender ∈ cb.Processes:
    cb.addRequest(sender, Retransmit(mid=this.mid))

Retransmit.fulfill(cb, owner):
  if cb.Wire[mid] != null:
    send({ owner }, cb.Wire[mid])

Retransmit.receive(cb, sender, payload):
  // internal use only, do nothing
```

Figure 3: Implementation of negative acknowledgment and retransmission

4.6 Satisfaction of Formal Properties

In this section, we provide proof sketches[1] that OldBlue satisfies the formal properties defined in Section 3.

THEOREM 1. *OldBlue satisfies Definition 3: Causal Consistency. I.e. no correct process p will execute* `CB.deliver(q, i, m)`*, until it has delivered all messages* $\mathtt{m}'$ *which causally precede* `m`.

PROOF SKETCH. The proof is by contradiction. Suppose that correct p delivers `m` before delivering a causally preceding message `m1`. Choose `m1` such that p has delivered a message `m2` where `m1` is the direct causal predecessor of `m2`. (By our assumption, such an `m1` and `m2` always exist with `m2` possibly equal to `m`.) Each message contains the ids of immediate causal predecessors, therefore `mid(m1)` ∈ `m2.parentids`. Thus, p delivered some message `m1'` ≠ `m1` with `mid(m1) == mid(m1')`. Therefore, `(m1, m1')` constitute a collision in `H()` contradicting its collision-resistance assumption and establishing the proof. □

We informally define and "associated message" to be retransmissions of a causal predecessors of `m` to p, or `LostMsg` requests for $\mathtt{m}' \rightarrow \mathtt{m}$ sent by p after receiving `m` to a correct process possessing $\mathtt{m}'$.

THEOREM 2. *OldBlue meets Definition 2: Validity. I.e. correct process p self-delivers its own messages and will deliver any message* `m` *from correct process q if the adversary delivers all associated messages.*

PROOF SKETCH. Self-delivery is immediate by the implementation of `CB.broadcast()`.

If p does not immediately `CB.deliver() m`, it is because some $\mathtt{m}' \rightarrow \mathtt{m}$ has not been received. p will issue a `LostMsg` request for $\mathtt{m}'$. Some correct process q holding $\mathtt{m}'$ will receive the request and eventually retransmit $\mathtt{m}'$ to p because the adversary is delivering all messages associated with `m`.This process continues until all causal predecessors of `m` have been received by p, at which time p will `CB.deliver() m`. □

THEOREM 3. *OldBlue meets Definition 4: Authenticity. I.e. a correct process q executes* `CB.deliver(p, i, m)` *at*

[1]Full proofs omitted from conference version due to space constraints.

most once for each value of `i` *(and any values of* p *and* `m`*) and, if* p *is correct, then* p *previously executed* `i = CB.broadcast(m)`*.*

PROOF SKETCH. If q were to execute `CB.deliver(p, i, m)` without p executing `i = CB.broadcast(m)`, it would constitute a successful attack on the provably secure Authenticated Encryption construct used by OldBlue.

For q to execute both `CB.deliver(_, i, m)` and `CB.deliver(_, i, m')` for $\texttt{m} \neq \texttt{m}'$ would entail a collision in `H()`, contradicting its collision-resistance assumption. □

4.7 Implementation Concerns

In the foregoing, we have abstracted away implementation tradeoffs by basing our design and arguments on idealized primitives. In this section, we identify desirable properties for practical implementations.

4.7.1 Limiting Process State

OldBlue can limit the amount of internal state each process must store by bounding the size of `Delivered`, `Undelivered`, and each queue in `Requests`.

The size of the delivered message set (`Delivered`) can be limited by garbage collecting stable messages. A message becomes stable only if the local process has delivered a causally newer message from each other process. This ensures the local process that no correct process will require retransmission of a stable message. If the length of a session and the maximum message transmission rate are bounded, an upper bound can be placed on the maximum number of unstable messages in a session.

The size of process request queues in `Requests` can be bounded as follows. The number of queued outgoing messages for the local process can be bounded by blocking the application once a predetermined threshold is reached. The number of outgoing retransmission requests can be bounded by generating them in a lazy fashion, at the moment that a retransmission request may be sent out. This provides an additional benefit of ensuring that outgoing retransmission requests reflect the most recent information available.

Request queues for remote processes can be bounded by fixing a parameter k and storing only k requests for each process. Limiting the number of retransmission requests in this way will not violate correctness because correct processes will continue to request missing messages until they are received.

The size of `Undelivered` can be limited by storing only k causally oldest messages from each process for some parameter k. Causal Consistency requires that, of all messages from a given correct process, the causally oldest message must be delivered first. In order to allow a process to determine the causal relationship between messages authored by the same process, all processes must be able to compute a consistent total ordering over the message ids of all messages sent by a given correct process. This can be achieved without impacting the collision-freeness of `mid()` by appending to the message digest a monotonically increasing sequence number indicating causality between messages authored by a single process.

4.7.2 Detecting Message Loss

Because the network is assumed to be unreliable and asynchronous, it is impossible to distinguish a lost message from a message with a long delivery delay. Therefore, processes must make an approximate determination of when a message has been lost and ensure that a `LostMsg` requests will eventually be issued for any message that has not been received. For efficiency, `LostMsg` delay strategies should ensure that sent `LostMsg` requests for messages that have not been lost will be infrequent and, when a message is lost, it should be discovered in a timely manner to minimize delivery delay of causally dependent messages.

This is often achieved in practice via timeouts. A timer begins when a message is received. If the timeout elapses before all causal predecessors are received, retransmission requests for missing predecessors will be sent. The timeout threshold can be optimized based on the observed delivery delay from each other process.

4.7.3 Minimizing Retransmission Requests

The simple retransmission mechanism given above pessimistically requests retransmission of every missing causal predecessor of each received message. Duplicate retransmission requests can be reduced by judiciously introducing delays before a retransmission request is issued—subject the requirement that *eventually* retransmission requests will be sent repeatedly for each lost message until the message is received.

To avoid feedback implosion, on expectation some small number of processes should request retransmission of a lost message. This can be achieved by randomly choosing retransmission request delays proportional to the number of processes and permitting a finite amount of additional backoff when requests from other processes are observed.

The number of retransmission requests can be further reduced by allowing them to identify entire missing causal subgraphs. Correct processes can choose their delay before requesting a retransmission in a way that causes the process missing the most messages to be the most likely to send their request first.

4.7.4 Minimizing Duplicate Retransmissions

The retransmission mechanism given above is also pessimistic. Every correct process will retransmit a requested message separately to each process that requested it. Ideally, we would like one correct process to retransmit (multicast if possible) a message to all processes that requested it and for correct processes to share the duty of retransmitting lost messages.

This can be achieved in a manner similar to reducing retransmission requests by randomly choosing retransmission delays. Deleting pending retransmission requests when a process indicates that it has delivered a causally newer messages than the requested message can help reduce duplicate retransmissions further.

5. EVALUATION

To better understand OldBlue's performance we created a protocol simulator using NS-3. We measured OldBlue's throughput and message delivery latency by sending null messages in sessions while varying parameters as indicated in Table 1. The results are shown in Figure 4.

For each parameter configuration, we simulated three runs of up to 5 minutes of protocol interaction with the following simplifying assumptions. Processes use the estimated round trip time (RTT) to other processes to delay `LostMsg` requests and retransmissions. Because estimation is not part of this

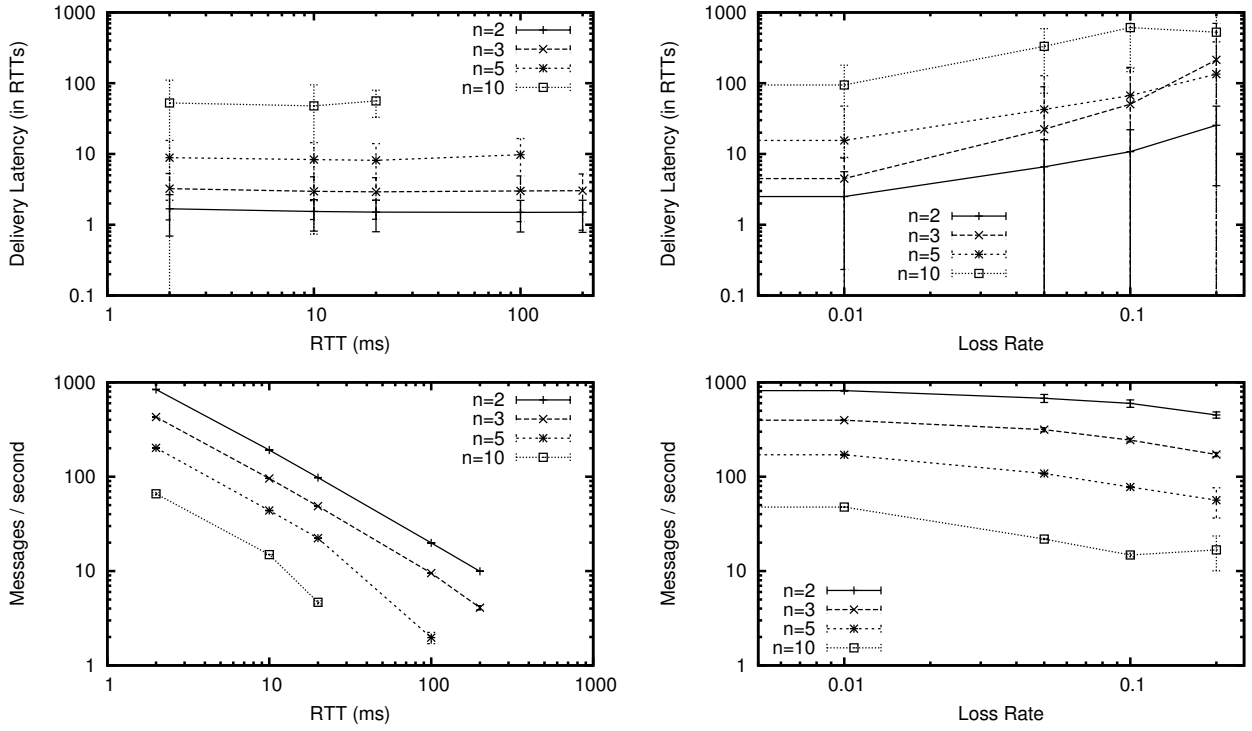

Figure 4: Null message delivery latency (top) and throughput (bottom) for varying RTT (left), loss rate (right) and session size.

Table 1: Simulation Parameter Choices

Number of processes	2, 3, 5, 10
Packet loss rate	0, 0.01, 0.05, 0.1, 0.2
Round Trip Time (ms)	2, 10, 20, 100, 200
Number of messages	1,000 or 5 minutes of sim. time

work, we fixed the estimated RTT to other processes at 4x the actual RTT. We conservatively estimated that each encryption or decryption operation took 4 microseconds and that each digital signature sign and verify operation took 480 microseconds. These values were obtained by running a microbenchmark that encrypts and signs 1 KB messages using AES-128 in CTR mode and RSASSA-PKCS1-v1_5 on our test system—an 8-core Intel Intel Xeon 2.67 GHz machine with 12 GB RAM running Ubuntu Linux 10.04 x86_64.

All session processes were locally connected to a simulated Ethernet LAN and ran OldBlue over UDP. Because we wanted to test the protocol without assuming hardware or IP multicast, simulated processes unicast messages to all other session processes. In contrast to the protocol above, when a process fulfills a `Retransmit` request, it multicasts the lost message to all processes.

Figure 4 depicts our measurements of the latency and throughput of null message delivery (respectively). In figures which vary RTT the network does not drop any packets. In figures which vary packet loss rate, the network RTT is fixed at 2 ms.

Message delivery latency is reported in multiples of the network RTT. The latency vs. RTT are ideal in the sense that, for fixed session size, even as RTT increases the latency remains close to a constant multiple of the RTT. The overhead should be able to be improved with an optimized implementation. The graph for latency vs. loss rate was obtained with the RTT fixed at 2 ms. As packet loss rate increases, latency increases.

Ideally, throughput should be unaffected by increasing RTT. However, the graphs of throughput vs. RTT show an approximate slope of -4 for the relationship between throughput and RTT. There are two factors that influence these results. When processes need to estimate the RTT to break synchrony, the simulator gives them a 4x overestimate of the network RTT. Thus, with an RTT estimation scheme that is more accurate in practice, the relationship between throughput and RTT can be improved. Furthermore, because all processes are connected to a simulated half-duplex Ethernet LAN, only one process may send a packet at a time. Thus, unlike real networks, increasing RTT causes a corresponding linear decrease in the effective bandwidth of the network connection. Therefore, this simulation depicts a pessimistic lower bound on throughput performance. An implementation on a more realistic network with an accurate RTT estimation strategy should show even stronger throughput performance.

Throughput also fairs well under packet loss when network characteristics are taken into account. Because the simulator simulates broadcast by n unicasts a lost message will create $2 \cdot n$ additional messages in the best case (n `LostMsg` requests and a retransmission to n processes). In most cases, the decrease in throughput is consistent with approximately $2 \cdot n$ additional messages being sent as the result of each message loss.

Examining the effect of session size on throughput and

latency we see that these performance metrics appear to decrease proportional to the square of the group size. This is unsurprising due to total amount of traffic between all processes growing proportionally to the square of the number of processes because of the use of direct unicast between all processes.

6. RELATED WORK

Broadcast protocols for distributed systems and mechanisms for preserving various ordering properties have a long history in the literature. Despite this, we are unaware of any protocol that provides the properties of OldBlue in a strong threat model. Many protocols [13, 1, 16, 9, 4] are not secure against Byzantine adversaries that can corrupt system members. Many protocols that are secure against a Byzantine adversary [10, 7, 19, 2, 11] either cannot ensure liveness among connected processes during a network partition or are not resilient to adversaries which can corrupt $t \geq \frac{n}{3}$ nodes as required by our threat model. Below, we compare concepts shared by most of these systems with OldBlue's requirements.

Psync. The causal broadcast property provided by OldBlue was partially inspired by Peterson et al.'s Psync protocol [16]. Psync provides an IPC mechanism for distributed systems that maintains the causal ordering between delivered messages in an environment subject to benign failures. OldBlue's Byzantine threat model leads to several nontrivial differences. Because Psync assumes that processes are correct, it is susceptible attack by corrupt processes. For instance, a duplicitous process can cause correct processes to deliver conflicting messages leading to differing causal histories. There is no mechanism in place to ensure that subsequent communication between the correct processes will disclose that their views differ. Psync does not attempt to enforce fairness. This allows corrupt processes to expend the resources of correct processes unchecked. Lastly, Psync's flow control mechanisms are stability-based — correct processes block when there are too many unstable messages. This has serious consequences to Psync's robustness to adversarial control because corrupt processes can effectively prevent any message from becoming stable, destroying session throughput for correct processes.

Reliable Broadcast. The properties of OldBlue are very similar to, and were inspired by, Reliable Broadcast. Reliable Broadcast provides a mechanism to broadcast messages such that all correct processes deliver the same set of messages and all correct processes deliver all messages broadcast by all correct processes. Hadzilacos and Toueg [10] demonstrate that Reliable Broadcast can be extended in a modular fashion to provide various guarantees on message delivery order.

A protocol provides Reliable Broadcast if it ensures the following four properties, as given by Cachin et al. [7]. In the following, each message is associated with a tag. The tag $ID.j.s$ is used to indicate the message with sequence number s sent by correct process P_j in session ID.

R-Validity If a correct process has r-broadcast $\mathtt{m}$ tagged with $ID.j.s$, then all correct processes r-deliver $\mathtt{m}$ tagged with $ID.j.s$, provided all correct processes have been activated on $ID.j.s$ and the adversary delivers all associated messages.

R-Consistency If some correct process r-delivers $\mathtt{m}$ tagged with $ID.j.s$ and another correct process r-delivers $\mathtt{m}'$ tagged with $ID.j.s$, then $\mathtt{m} = \mathtt{m}'$.

R-Totality If some correct process r-delivers a message tagged with $ID.j.s$, then all correct processes r-deliver some message tagged with $ID.j.s$, provided all correct processes have been activated on $ID.j.s$ and the adversary delivers all associated messages.

R-Authenticity For all ID, senders P_j, and sequence numbers s, every correct process r-delivers at most one message $\mathtt{m}$ tagged with $ID.j.s$. Moreover, if P_j is correct, then $\mathtt{m}$ was previously r-broadcast by P_j with sequence number s.

The definition of OldBlue purposefully differs from Reliable Broadcast in the following ways:

1. R-Validity requires a correct process to r-deliver a message only if the adversary delivers all associated messages. By contrast, OldBlue's Validity property requires liveness among processes in a connected component during a network partition by requiring any q that receives all messages associated with `CB.broadcast(m)` to deliver $\mathtt{m}$.

2. R-Totality ensures that all correct processes deliver the same set of messages regardless of the correctness of the author. This requires communication between correct processes for each delivery operation. Cachin et al. note that R-Totality is responsible for the $O(n^2)$ communication complexity of most Reliable Broadcast protocols.

   R-Totality conflicts with OldBlue's requirement for liveness during network partitions. Causal Consistency (Definition 3) is weaker than R-Totality. Instead of guaranteeing consistency between *all* correct processes, Causal Consistency guarantees consistency pairwise. Upon delivery of message $\mathtt{m}$ from a correct sender, a correct deliverer is guaranteed to be consistent with the sender with respect to $\mathtt{m}$ and causally preceding messages.

3. R-Consistency and R-Authenticity are both guaranteed by Authenticity (Definition 4).

Consistent Broadcast. A Consistent Broadcast [7] protocol satisfies R-Validity, R-Consistency, and R-Authenticity above but *not* R-Totality. As noted above, R-Validity conflicts with our requirements. Therefore, Consistent Broadcast is unsuitable for our purposes as well.

Byzantine Agreement. Byzantine Agreement, the Byzantine Generals' Problem [12], and the closely related Consensus [10] Problem, refer to the problem of ensuring that all processes in a distributed system agree on a value proposed by one, or more, processes. Like Reliable Broadcast, any protocol which depends on Byzantine Agreement is incompatible with OldBlue's formal requirements. Byzantine Agreement requires agreement — if a correct process decides x, all correct processes eventually decide x and, if the

proposing process was correct, x was the value proposed. Unless all messages are known a priori, this property is incompatible with Validity—processes in a connected component must be able to deliver messages even during a network partition. For Byzantine Agreement to satisfy Validity, processes in different components of a network partition would need to reach the same session transcript without communicating with processes in other components.

7. FUTURE WORK

We are currently simulating various aspects of the OldBlue protocol to help guide implementation decisions. We plan to provide an implementation for general use. We also intend to explore membership algorithms compatible with our threat model.

8. CONCLUSION

We have presented OldBlue—a causal broadcast protocol secure against an arbitrary number of Byzantine failures. We have formally defined the properties that OldBlue provides and proven that the proposed protocol meets the formal definitions.

9. REFERENCES

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *Proceedings of the Twenty Second International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[2] M. Backes and C. Cachin. Reliable broadcast in a computational hybrid model with byzantine faults, crashes, and recoveries. In *Proc. of Intl. Conference on Dependable Systems and Networks*, pages 37–46, 2003.

[3] M. Bellare and C. Namprempre. Authenticated Encryption: Relations Among Notions and Analysis of the Generic Composition Paradigm. *J. Cryptol.*, 21(4):469–491, Sept. 2008.

[4] K. P. Birman and T. A. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

[5] J.-M. Bohli, M. I. González Vasco, and R. Steinwandt. Secure Group Key Establishment Revisited. *Int. J. Inf. Secur.*, 6(4):243–254, June 2007.

[6] G. Bracha and S. Toueg. Asynchronous Consensus and Broadcast Protocols. *J. ACM*, 32(4):824–840, Oct. 1985.

[7] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and Efficient Asynchronous Broadcast Protocols. Report 2001/006, Cryptology ePrint Archive, 2001. http://eprint.iacr.org/2001/006.

[8] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of Operating Systems Design & Implementation (OSDI)*, pages 173–186, Berkeley, CA, USA, 1999.

[9] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997.

[10] V. Hadzilacos and S. Toueg. A Modular Approach to the Specification and Implementation of Fault-Tolerant Broadcasts. Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca, NY, USA, May 1994. http://www.cs.toronto.edu/~vassos/research/publications/HT94/paper.ps.gz.

[11] K. Kursawe and V. Shoup. Optimistic Asynchronous Atomic Broadcast. In *Proceedings of International Colloqium on Automata, Languages and Programming*, LNCS 3580. Springer, 2001. Full version. Revised April 19, 2002. http://www.shoup.net/papers/ks.pdf.

[12] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[13] P. Melliar-Smith, L. Moser, and V. Agrawala. Broadcast Protocols for Distributed Systems. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):17–25, 1990.

[14] A. Mislove, G. Oberoi, A. Post, C. Reis, P. Druschel, and D. S. Wallach. AP3: Cooperative, decentralized anonymous communication. In *Proceedings of ACM SIGOPS European workshop*, EW 11, Leuven, Belgium, 2004.

[15] A. Nambiar and M. Wright. Salsa: A Structured Approach to Large-Scale Anonymity. In *Proceedings of ACM Conference on Computer and Communications Security*, New York, NY, USA, Oct. 2006.

[16] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and Using Context Information in Interprocess Communication. *ACM Trans. Comput. Syst.*, 7(3):217–246, 1989.

[17] J. Reardon, A. Kligman, B. Agala, and I. Goldberg. KleeQ: Asynchronous Key Management for Dynamic Ad-Hoc Networks. Technical Report CACR 2007-03, Center for Applied Cryptographic Research, University of Waterloo, Waterloo, ON, Canada, 2007. http://cacr.uwaterloo.ca/techreports/2007/cacr2007-03.pdf.

[18] M. Reiter and L. Gong. Preventing Denial and Forgery of Causal Relationships in Distributed Systems. In *IEEE Symposium on Security and Privacy*, pages 30–40, Berkeley, CA, USA, 1993.

[19] M. K. Reiter. The Rampart Toolkit for Building High-Integrity Services. In *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*, pages 99–110, London, UK, 1995. Springer-Verlag.

[20] S. W. Smith and J. D. Tygar. Signed Vector Timestamps: A Secure Protocol for Partial Order Time. Technical Report CMU-CS-93-116, Department of Computer Science, Carnegie Mellon University, 1993. http://reports-archive.adm.cs.cmu.edu/anon/1993/CMU-CS-93-116.ps.

[21] S. W. Smith and J. D. Tygar. Security and Privacy for Parital Order Time. In *Proceedings of the Parallel and Distributed Computing Systems Conference*, pages 70–79, Oct. 1994.

[22] P. Wang, I. Osipkov, N. Hopper, and Y. Kim. Myrmic: Provably secure and efficient DHT routing. Technical Report 2006/20, Digital Technology Center, University of Minnesota, 2006. http://www.dtc.umn.edu/publications/reports/2006_20.pdf.