GROUP COMMUNICATION PROTOCOLS AND A FRAMEWORK FOR INTRUSION-TOLERANT DISTRIBUTED APPLICATIONS[*]

HariGovind V. Ramasamy
University of Illinois, Urbana, IL 61801, USA
ramasamy@crhc.uiuc.edu

Abstract

We present an overview of a suite of protocols for *practical* intrusion-tolerant group communication. The protocols are very efficient in the normal case of no faults, are reasonably efficient even when faults occur, provide strong reliability guarantees, and operate under realistic timing assumptions. The protocol suite will be implemented in a reusable, reconfigurable, and portable framework called CoBFIT. The framework contains several support features for intrusion tolerance so that it can serve as a foundation for building and evaluating various design choices for intrusion-tolerant group communication protocols and group-communication-based applications. The protocol suite and the CoBFIT framework will constitute the CoBFIT toolkit. The eventual goal is to develop a rich set of group communication services in the CoBFIT toolkit that can be used for embedding intrusion tolerance in several distributed applications.

1. Introduction

Intrusion tolerance is an approach for handling malicious attacks into computer systems. In this approach, the impracticability of defending against all attacks is recognized and intrusions are expected, but the system is expected to provide proper service in spite of them (possibly in a degraded mode). Intrusion tolerance when combined with traditional security, can constitute "defense-in-depth," making the attacker expend more effort before bringing the system down.

Group communication is a key building block for many intrusion- and fault-tolerant distributed systems. Group communication systems (GCSs) have been developed to ensure state consistency among the processes that constitute a group in the presence of failures. While a considerable amount of work has been done to make GCSs tolerate benign failures (see [5] for a comprehensive survey), the problem of building GCSs that can tolerate the malicious corruption of some group members has been gaining momentum only more recently

[*]This research has been supported by DARPA contract F30602-00-C-0172.

(e.g., Rampart Security toolkit [16], SecureRing [13], ITUA GCS [15], and Correia's GCS [6]). Our previous work [15] has shown that the overhead for tolerating malicious faults in such intrusion-tolerant GCSs is significant (even when there are no actual faults), compared to the overhead for tolerating just benign faults, like crashes. While Correia's intrusion-tolerant GCS [6] shows better performance than other implementations and its idea of implementing group communication protocols using a trusted timely computing base (TTCB) is novel, the number of environments in which such a TTCB can be implemented is limited (because it assumes a synchronous system model). Recent work by Castro and Liskov in [4] has shown that Byzantine fault-tolerant state machine replication can be made practical with modest latencies; however, their system is not a GCS. Furthermore, existing intrusion-tolerant GCSs cannot operate in WAN environments because of reliance on synchrony assumptions for correctness. They must remove faulty members from the group to make progress, and hence rely on failure detectors that cannot be accurate in asynchronous environments. SINTRA [3] provides intrusion-tolerant replication on the Internet, but lacks a group membership protocol and supports only static groups; hence, it is not a GCS. Though there has been research on wide-area GCSs that tolerate benign faults (e.g., [2]), the problem of developing wide-area, intrusion-tolerant GCSs remains largely unaddressed.

To address the above issues , we will develop two suites of intrusion-tolerant group communication protocols: (1) a suite of protocols that incur minimal overhead in the fault-free case, are efficient even when faults occur, provide strong reliability guarantees, and operate under partial synchrony assumptions [9], and (2) a suite of protocols for wide area networks that can be used for making critical services distributed over the Internet intrusion-tolerant (e.g., certification authorities, directory services, and backbone routers). The first suite of protocols is aimed towards obtaining significant performance improvements (while retaining strong guarantees and operating under realistic timing assumptions) compared to existing intrusion-tolerant GCSs. The second is aimed towards making intrusion-tolerant group communication relevant to today's Internet-based services and making those services tolerant to insider attacks. Thus, our overall goal is to make intrusion-tolerant group communication (1) practical, by reducing its inefficiencies and high costs, and (2) applicable to today's Internet-based critical distributed services.

Together, the two protocol suites present a diverse set of application requirements, operating environments, and system configurations. However, the implementation of both protocol suites requires several common support features, such as event handling, secure buffer management, cryptography, and coordination of inputs from multiple intrusion detection mechanisms. Such features are commonly needed in developing not only intrusion-tolerant GCSs but several other intrusion-tolerant systems (e.g., [4][8]). This suggests that if the support could be isolated (rather than dispersed throughout the system), it could form the basis of a reusable framework for easily constructing and testing a variety of intrusion-tolerant applications. We separate this commonly needed support for intrusion tolerance from the intrusion-tolerant protocols, with the aim of implementing the support features in a reusable, reconfigurable, and portable framework, called "CoBFIT - a Component-Based Framework for Intrusion Tolerance." Implementing the two protocol suites within the CoBFIT framework will yield the *CoBFIT toolkit* with a rich set of group communication services that can be used for embedding intrusion tolerance in several distributed applications.

The focus of this paper is the first suite of protocols and the CoBFIT framework. We present an overview of our approach in designing the first suite of protocols (Section 2). We also briefly describe the main components of the CoBFIT framework, and its design and implementation principles (Section 3). Finally, we conclude by summarizing the intended contributions of this research, current status, and future work (Section 4).

2. Practical Intrusion-Tolerant Group Communication

Any GCS has two fundamental services: a group membership service and a multicast service. The group membership service is responsible for adding members to the group, removing members from the group, and maintaining the correct membership list at all correct processes. The basic multicast service offered by GCSs is reliable multicast, which ensures that all correct processes deliver the same set of messages but does not provide any message ordering guarantees. Variants of reliable multicast have been developed that provide FIFO, causal, or total order guarantees or a combination of those guarantees.

The main idea behind our first suite of protocols is based on two observations: (1) at the core of many group communication protocols is a consensus algorithm, and (2) process failures are the exception rather than the norm. We leverage these observations by creating a Byzantine-fault-tolerant consensus algorithm that is very efficient in the fault-free case (and reasonably efficient even when faults occur), and developing group communication protocols with consensus as the building block. (Previously, [11] used consensus to build group communication services, but assumed only benign crash faults. Recently, Yin et al. [18] separated agreement from the execution of Byzantine-fault-tolerant state-machine replication, but their system is not a GCS.)

In [14], we describe our consensus algorithm, called Lazy Byzantine Consensus (LBC) which operates under partial synchrony assumptions [9][12]. In standard consensus, each process starts with an initial value. LBC is a generalization of standard consensus in which only a subset of processes, called the *primary committee* (pc), start with an initial value. The pc consists of $t + 1$

processes, where t is the maximum number of replicas that could be Byzantine-faulty out of $n \geq 3t+1$ replicas $\{p_1, p_2, \cdots, p_n\}$. Only the processes in the pc exchange messages and try to reach a decision. This means that in the normal case of no faults, only $t+1$ out of $n \geq 3t+1$ processes generate messages and perform the processing to reach consensus. This contrasts with other consensus algorithms in which all correct processes generate messages and try to reach consensus. (Even in coordinator-based algorithms like [12], processes other than the coordinator have to send their estimate values and exchange messages with the coordinator in order to reach consensus.) Our consensus algorithm uses the minimum number of communication steps (two) required to solve consensus in fault-free runs and requires only the minimum number of processes necessary to reach consensus ($n = 3t+1$). When the primary committee is unable to reach a decision, a reselection of the committee occurs. The efficiency of the algorithm in the presence of faults then depends on the number of reselections before a decision can be reached. With the optimizations described in [14], the number of reselections is upper-bounded by $2t+1$ or $O(t)$. If the number of actual faults f is less than the maximum number of allowed faults t, then (in the optimized version) the number of reselections is upper-bounded by $O(f)$ (much as in early stopping algorithms by Dolev et al. [7]). The interested reader is referred to [14] for a detailed description of the algorithm, complete with efficiency analysis and proofs of correctness.

We now briefly summarize the design of our first suite of intrusion-tolerant group communication protocols using the LBC consensus algorithm.

Group Membership: Each group member p_k maintains two lists: *new-list* and *accused-list*. The *new-list* contains a list of processes that p_k wants to be added to the group; the *accused-list* contains a list of processes that p_k wants to be removed from the group. A group member p_i is added to p_k's *accused-list* only after p_k has learned through digitally signed *suspect* messages that at least two-thirds of the group members suspect p_i to be faulty. Similarly, a non-member p_j is added to p_k's *new-list* only after p_k has learned through digitally signed *approve* messages that at least two-thirds of the group members approve of p_j's addition to the group. The group membership protocol uses the LBC algorithm to ensure that the next membership lists (*view*s) installed at all correct members are the same. The initial value for consensus at a process p_k is the next view of the group, obtained by removing the *accused-list* members from the current view, and adding the *new-list* members to the current view. In order for the protocol to progress, the intersection of the current view and next view should contain at least $2t+1$ correct members (assuming that a process at the time of joining the group is correct, which can be ensured by admission control policies). During agreement on the next view (at the LBC algorithm level), if one or more of the pc members are among the list of processes to be removed from the next view (i.e., some pc member finds that a fellow pc member is in its

accused-list), then pc reselection will be initiated. The reselection will repeat until none of the pc members have fellow pc members in their *accused-list*s.
Reliable Multicast: A reliable multicast protocol guarantees that a message transmitted by a correct sender is delivered by all correct processes. If the sender is malicious, then either the message is delivered with the same contents at all correct processes or no correct process delivers the message. When a new view is installed, the reliable multicast protocol at a group member p_k creates one instance of the LBC algorithm for every other group member p_i. The instance created thus for p_i persists for the duration of the view and tries to reach agreement on the contents of the next message m from p_i to be delivered at p_k. The initial value for consensus is the hash of the message m. After agreement on the contents of m, a correct process can deliver m. Standard buffering and negative acknowledgment techniques are used to deal with message losses.
View Synchronous Multicast: Informally stated, a view synchronous multicast protocol ensures that all correct processes deliver a multicast message m in the same view. During a new view installation, the protocol ensures that the set of multicast messages delivered at all correct processes that are in $V_x \cap V_{x+1}$ (where V_x denotes the current view and V_{x+1} denotes the next view) is the same for view V_x. All processes in $V_x \cap V_{x+1}$ exchange their *stable set*s after the group membership protocol at those processes has agreed on V_{x+1}; process p_k's stable set indicates the highest-sequence-numbered multicast message from every group member that has been delivered at p_k. The stable sets received at a correct process p_k from other processes are forwarded to the group. Since the messages are signed by the sender, it is easy to detect a process that lies about its stable set and sends different stable sets to different members. Such a detection will cause the group membership protocol to revise the next view V_{x+1} by excluding the newly detected faulty process from V_{x+1} (after invoking the LBC algorithm to reach agreement on the revised next view). After the initial exchange of stable sets among all the group members, the view synchronous multicast protocol invokes the LBC algorithm to reach agreement on the highest-sequence-numbered message from every member of V_x delivered at any process in $V_x \cap V_{x+1}$. The initial value for consensus would be a set of ordered pairs $\{(sn_1, p_{sn_1}), \cdots, (sn_{|V_x|}, p_{sn_{|V_x|}})\}$. The i^{th} element (sn_i, p_{sn_i}) indicates the highest sequence number sn_i of all multicasts received from process $p_i \in V_x$ by any process in $V_x \cap V_{x+1}$ and the process p_{sn_i} ($p_{sn_i} \in V_x \cap V_{x+1}$) that claims to have received that sequence-numbered multicast. After the consensus algorithm terminates, if a process $p_k \in V_x \cap V_{x+1}$ has not yet delivered messages up to sequence number sn_i from process p_i, then p_k asks process p_{sn_i} to retransmit the missing messages. If p_{sn_i} refuses to retransmit the missing messages, then p_k will send a request for retransmission of the missing messages from p_i to all processes in $V_x \cap V_{x+1}$. If no process

in $V_x \cap V_{x+1}$ has delivered the missing messages, then it is clear that p_{sn_i} lied about the existence of a multicast message from p_i with sequence number sn_i. In that case, the group membership will revise the next view V_{x+1} to exclude p_{sn_i} from V_{x+1} (after agreement using the LBC algorithm), and the view synchronous multicast protocol will repeat the steps outlined above.

Atomic Multicast: We obtain atomic multicast by implementing a total ordering protocol on top of the reliable multicast protocol. A total ordering protocol guarantees that if two correct processes both deliver messages m_1 and m_2, then they deliver them in the same order. Our total ordering protocol is an adaptive variant of the protocol we described in [15], and uses the LBC algorithm to perform the adaptation. In [15], we described a total ordering protocol in which sequence numbers assigned to processes are globally unique. Whenever the group membership changes, the set of all possible sequence numbers is partitioned, and each group member is assigned a partition. At view installation time, each group member p_i is associated with an initial sequence number seq_orig_i and a monotonically increasing sequence-number-generating function gf_i. The set of sequence numbers that can be generated by process p_i is given by $\{seq\_orig_i, gf_i(seq\_orig_i), gf_i(gf_i(seq\_orig_i)), \cdots\}$. Messages are delivered in the order of their global sequence numbers. If a process does not send a message with a particular sequence number, it can stall the delivery of greater-sequence-numbered messages from other processes. To prevent this, processes are required to send protocol-level *null* messages if they don't have any other messages to send. The progress of the protocol is monitored, and any process that stalls the progress of the protocol will be reported to the group membership protocol for removal from the group. The efficiency of the protocol depends on how closely the sequence-number-generating functions model the actual message traffic pattern. The protocol is efficient when the group has a predictable message traffic pattern, but suffers from low performance (because of the *null* messages) when the traffic pattern fluctuates. Our new total ordering protocol adapts to such fluctuations by revising the generating functions if the $\frac{throughput\ due\ to\ null\ messages}{total\ throughput}$ goes above a threshold and stays above the threshold for a sufficiently long time (the latter condition avoids repeated revisions of the generating functions due to transient traffic pattern fluctuations). The revision of the generating functions can be initiated by any member; it is followed by the invocation of the LBC algorithm by the group members to reach agreement on the proposed revision to the generating functions. The decision reached after the execution of the LBC algorithm either approves or disapproves of the proposed revision, depending on how closely the proposed revision models the recent message traffic history. If a revision is approved, the group members adopt the new generating functions; if it is disapproved, then the members continue with the old generating functions.

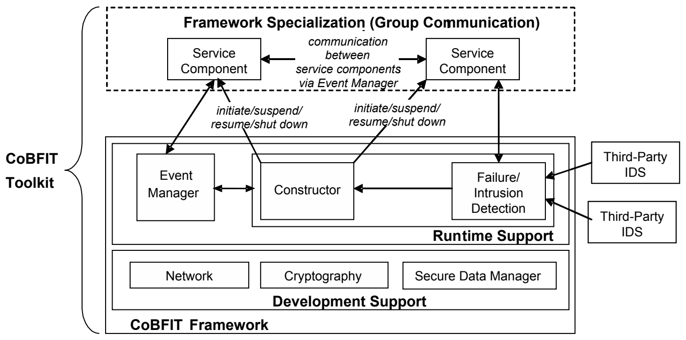
Framework Specialization (Group Communication)
Service Component
communication between service components via Event Manager
Service Component
initiate/suspend/ resume/shut down
initiate/suspend/ resume/shut down
CoBFIT Toolkit
Event Manager
Constructor
Failure/ Intrusion Detection
Third-Party IDS
Third-Party IDS
Runtime Support
Network
Cryptography
Secure Data Manager
Development Support
CoBFIT Framework

Figure 1. The CoBFIT Toolkit

3. The CoBFIT Framework

Figure 1 shows the CoBFIT toolkit which consists of a set of group communication services built using the CoBFIT framework. Both the framework and services will be realized by a collection of components. A CoBFIT component will be a coherent, encapsulated part of the CoBFIT toolkit that provides its functionality in the form of clearly defined object and event interfaces that other components can use. Framework components will implement the abstractions and primitives that are commonly needed in intrusion-tolerant systems. Some framework components provide development support (i.e., they are useful in the development of CoBFIT service components), while others provide run-time support. A service component will be the implementation of a protocol or algorithm (in our case, a group communication protocol) that provides a set of desired properties. We will now describe the design of the framework components through which the service components operate.

Event Manager: Event processing is based on a publish-subscribe model. The service components publish the events they generate to the Event Manager. They also subscribe to the events that they are interested in receiving (handling) from the Event Manager. Thus, at initialization, the Event Manager will know the set of all events, and the handlers bound to them. These bindings can change dynamically when one or more of the service components are replaced or reconfigured during run-time. The Event Manager component is based on the Reactor Software pattern [17] for event handling. The Event Manager will detect and demultiplex any event in the CoBFIT toolkit and dispatch it to the service components that have subscribed to that event.

Constructor: This component is responsible for the reconfigurability of the CoBFIT toolkit and is based on the Component Configurator design pattern [17]. All service components will implement a uniform component management interface through which they can be configured and controlled by the Constructor. The interface will define operations by which a component can be (re)initialized, shut down, and made to suspend/resume execution. The Constructor will maintain a component repository. It will implement a mechanism that interprets and executes a script specifying (at run-time) which of the available components to link into and unlink out of the CoBFIT toolkit.

Secure Data Manager: This component will provide primitives and data structures that facilitate secure and efficient data management. It will provide "safe classes" whose methods are essentially wrappers around C/C++ standard library functions that are generally perceived to be unsafe, because of their high susceptibility to buffer overflows and format string problems. It will also provide classes to manage messages efficiently (for example, using a single reference-counted message copy instead of multiple copies) with operations for buffering, marshalling and demarshalling, fragmenting and reassembling, and reordering messages received out of sequence. The message operations will be made safe through sanity checks made wherever necessary (e.g., bounds checks on array and pointer references, or a message format check on received messages).

Cryptography: This component will facilitate deployment of cryptography in the service components by providing a uniform way to access commonly available third-party cryptographic libraries. In essence, it will define interfaces for common cryptographic operations (e.g., signing/verifying a message, encrypting/decrypting a message, or computing the digest for a message) that could be invoked by the service components and will adapt the interface of the chosen cryptographic library to the defined interface. Since service components do not make direct calls to the third-party cryptographic library, but instead invoke interfaces in the Cryptography component, the components could be reused even if the choice of cryptographic library is changed later.

Network: All CoBFIT service components that want to send messages to and receive messages from the network will do so through the Network component. The Network component is based on the Wrapper Facade design pattern [10] and will consist of classes that encapsulate platform-specific low-level network functions and data within a type-safe, portable, object-oriented interface. The component will hide the underlying transport mechanism from the service components, thereby providing the flexibility to change the transport mechanism without having to modify the service components.

Failure/Intrusion Detection: Intrusion detection for CoBFIT service components could be internal, external, or both. Internal intrusion detection will be done by any service component based on observed anomalies or deviations from the specifications. External intrusion detection will employ third-party intrusion detection systems (IDSs). Many of these IDSs are constantly updated based on new attacks. The Failure/Intrusion Detection component enforces a clean separation between intrusion detection and intrusion response mechanisms. It allows the service components to be independent of the specific intrusion detection tools or mechanisms used. The component is based on the Mediator software pattern [10] and will act as the hub of communication for intrusion detection. It will serve as the central sink for intrusion detection reports or suspect reports from internal or external intrusion detection sources. It will process reports from diverse sources and implement policies to determine which reports should actually lead to system adaptation. For such reports, it will generate a failure detection event to which CoBFIT components can subscribe; the components can then respond to the event based on the particular protocol or strategy that they implement.

The CoBFIT framework presented above is intended to be robust and portable. While we believe that it is impossible to build the framework to be entirely free of vulnerabilities, our goal is to minimize them. As described above, the design of CoBFIT framework components is based on software patterns [10][17] that have been well-documented and scrutinized in the literature by expert software practitioners. We are currently implementing the CoBFIT framework using the Adaptive Computing Environment (ACE) toolkit [1]. The use of the ACE toolkit for implementing CoBFIT would facilitate portability of the framework, since ACE provides a rich set of reusable C++ wrapper facades and framework components that perform common communication software tasks across a range of various operating systems and platforms.

4. Conclusion: Summary and Future Work

In summary, this research aims to develop the CoBFIT toolkit which will consist of two suites of intrusion-tolerant group communication protocols and a reusable framework that implements support features commonly needed for intrusion tolerance. The two protocol suites aim to make intrusion-tolerant group communication (1) practical, by reducing its inefficiencies and high costs, and (2) applicable to today's Internet-based critical distributed services.

We presented an overview of our approach to practical intrusion-tolerant group communication using Lazy Byzantine Consensus as the building block. Our LBC algorithm is very efficient (in that it uses less processing power and message exchanges compared to other algorithms) in the fault-free case, has good performance even in the presence of faults, and exhibits worst-case overhead proportional to the actual number of faults in the system. We expect the group communication protocols built using the LBC algorithm to exhibit similar traits.

We also presented an overview of the CoBFIT framework, which provides specialized support commonly needed for intrusion-tolerant systems, such as coordination of inputs from multiple intrusion detection sources, cryptography, and safe buffer management. The framework also has components for event management, reconfiguration management, and network I/O that would enable services built using the framework to be reconfigurable, adaptable, and portable.

Currently, we are implementing the first suite of protocols and the CoBFIT framework using the ACE toolkit. Future work includes (1) addition of more support mechanisms for intrusion tolerance to the framework (e.g., support for replication of services on multiple nodes and dynamic management of replicas) and (2) design of the second suite of WAN-oriented protocols.

Acknowledgments

We thank Prof. William H. Sanders and Dr. Adnan Agbaria for their valuable input and comments, and Jenny Applequist for her editorial assistance.

References

[1] The ADAPTIVE Computing Environment. http://www.cs.wustl.edu/~schmidt/ACE.html.

[2] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. Technical Report CNDS 98-4, Johns Hopkins University, 1998.

[3] C. Cachin and J. A. Poritz. Secure Intrusion-Tolerant Replication on the Internet. In *Proc. Intl. Conf. on Dependable Systems and Networks (DSN-2002)*, pages 167–176, 2002.

[4] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proc. 3rd Symp. on Operating Systems Design and Implementation (OSDI-99)*, pages 173–186, February 1999.

[5] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 33(4):1–43, December 2001.

[6] M. Correia. *Intrusion Tolerance Based on Architectural Hybridization*. PhD thesis, University of Lisbon, 2003.

[7] D. Dolev, R. Reischuk, and H. R. Strong. Early Stopping in Byzantine Agreement. *Journal of the ACM*, 37(4):720–741, October 1990.

[8] B. Dutertre, V. Crettaz, and V. Stavridou. Intrusion-Tolerant Enclaves. In *Proc. IEEE Intl. Symp. on Security and Privacy*, pages 216–224, 2002.

[9] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, April 1988.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2001.

[11] R. Guerraoui and A. Schiper. The Generic Consensus Service. *IEEE Trans. on Software Engineering*, 27(1):29–41, January 2001.

[12] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Solving Consensus in a Byzantine Environment Using an Unreliable Fault Detector. In *Proc. Intl. Conf. on Principles of Distributed Systems (OPODIS)*, pages 61–75, December 1997.

[13] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing Protocols for Securing Group Communication. In *Proc. 31st Annual Hawaii Intl. Conf. on System Sciences (HICSS)*, volume 3, pages 317–326, January 1998.

[14] H. V. Ramasamy, A. Agbaria, and W. H. Sanders. Semi-Passive Replication in the Presence of Byzantine Faults. Technical Report UILU-ENG-04-2202, Center for Reliable and High-Performance Computing, University of Illinois, Urbana-Champaign, February 2004.

[15] H. V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders. Quantifying the Cost of Providing Intrusion Tolerance in Group Communication Systems. In *Proc. Intl. Conf. on Dependable Systems and Networks (DSN-2002)*, pages 229–238, 2002.

[16] M. K Reiter. The Rampart Toolkit for Building High-Integrity Services. *Theory and Practice in Distributed Systems (LNCS 938)*, pages 99–110, 1995.

[17] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley, 2001.

[18] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proc. 19th ACM Symp. on Operating Systems Principles (SOSP-2003)*, pages 253–267, 2003.