Byzantine Fault Isolation in the Farsite Distributed File System

John R. Douceur and Jon Howell
Microsoft Research
{johndo, howell}@microsoft.com

ABSTRACT

In a peer-to-peer system of interacting Byzantine-fault-tolerant replicated-state-machine groups, as system scale increases, so does the probability that a group will manifest a fault. If no steps are taken to prevent faults from spreading among groups, a single fault can result in total system failure. To address this problem, we introduce Byzantine Fault Isolation (BFI), a technique that enables a distributed system to operate with application-defined partial correctness when some of its constituent groups are faulty. We quantify BFI's benefit and describe its use in Farsite, a peer-to-peer file system designed to scale to 100,000 machines.

1 INTRODUCTION

Farsite [2] is a distributed peer-to-peer file system that runs on a network of desktop workstations and provides centralized file-system service. File storage in Farsite is secure, even though the system runs on unsecured machines. This security is established, in part, through the use of Byzantine Fault Tolerance (BFT), a well-known mechanism for building trusted services from untrusted components [14, 6]. BFT enables a service to continue functioning correctly as long as fewer than a third of the machines it is running on are faulty.

Farsite is designed to be scalable. As more and more workstations make use of the file-system service, the resources of these workstations become available for use by the system. However, the BFT technique cannot exploit these additional resources to achieve the greater throughput demanded by an increasing user base: Adding machines to a BFT group decreases throughput, rather than increasing it.

To achieve an increase in throughput with scale, Farsite partitions its workload among multiple interacting BFT groups. Unfortunately, as the count of BFT groups increases, so too does the probability that some group will contain enough faulty machines that the group will be unable to suppress the fault. If the system design does not account for the failure of one or more BFT groups, a single group failure can cause the entire system to fail.

The alternative to total failure is degraded operation, wherein individual group failures cause the system to operate in a way that is still partially correct, rather than completely unspecified. However, "partial correctness" is not something that can be defined in an application-independent fashion [11]. It is thus not possible to build a generic substrate that enables an arbitrary service to degrade gracefully in a meaningful and useful manner.

We have therefore developed a methodology for designing a distributed system of BFT groups, wherein a faulty group is prevented from corrupting the entire system. We call our method Byzantine Fault Isolation (BFI). BFI makes use of formal specification [15] to constrain the semantic behavior of a faulty system. The technique involves selectively weakening the system semantics and concomitantly strengthening the system design. Formal specification helps determine when these two processes have satisfactorily converged.

The next section surveys previous approaches to resisting Byzantine faults. Section 3 describes the Farsite file system, the context for our work. Section 4 quantifies the value of isolating Byzantine faults, and Section 5 describes our technique. Section 6 shows an example of BFI in Farsite, and Section 7 summarizes.

2 PREVIOUS WORK

In 1980, Pease et al. [23] introduced the problem of reaching agreement among a group of correctly functioning processors in the presence of arbitrarily faulty processors; they proved that a minimum of $3t + 1$ processors are needed to tolerate t faults. Two years later, they christened this the *Byzantine Generals Problem* [14], as it has been known ever since. A large body of research has addressed the problem of Byzantine agreement, the first decade of which is well surveyed by Barborak and Malek [4].

In the mid-to-late 1990's, several researchers combined Byzantine-fault-tolerant protocols with state-machine replication [26] to produce toolkits such as Rampart [24], SecureRing [12], and BFT [6]. These toolkits provide a replication substrate for services written as deterministic state machines. The substrate guarantees that the service will operate correctly as long as fewer than a third of its replicas are faulty. An unfortunate property of these toolkits is that their throughput scales negatively: As the group size grows, the system throughput shrinks, which is the exact opposite of the behavior desired for scalable systems.

Throughput scaling is non-positive because a non-decreasing fraction of replicas redundantly perform each computation. Throughput scaling is made negative by the message load of each processor, which is linear in group size. Some recent research has addressed the latter issue: Lewis and Saia [16] have developed a protocol that probabilistically reaches agreement if fewer than an eighth of the replicas are faulty. The message workload

of each processor is logarithmic in group size, so throughput scaling is less dramatically negative than for BFT. Abd-El-Malek et al. [1] have built a replicated state machine based on the Query/Update (Q/U) protocol, which requires $5t + 1$ processors to tolerate t Byzantine faults. In theory, this protocol has zero throughput scaling with system size; however, their implementation exhibits negative throughput scaling, albeit at a lower rate than BFT.

The above systems all exhibit two properties: (1) non-positive throughput scaling and (2) all-or-nothing failure semantics, meaning that failures beyond the tolerated threshold can cause the entire system to fail.

In the absence of a Byzantine-fault-tolerant substrate that provides positive throughput scaling, researchers have built systems that partition their workload among multiple machines. However, as the system size grows, so does the expected number of faulty machines, which in turn – given all-or-nothing failure semantics – leads to an increasing likelihood of total system failure. We observe that this problem could be assuaged if there were some means to limit the spread of Byzantine faults.

Three avenues of research are related to this problem: First, several researchers have isolated Byzantine faults in distributed problems of academic interest, such as dining philosophers [21], vertex coloring [21], and edge coloring [25, 18]. Their solutions employ self-stabilizing protocols to guarantee that correct results are eventually obtained by all nodes that are beyond a specified distance (the “containment radius”) from faulty nodes. The formal notion of *fault containment* for self-stabilizing systems was introduced by Ghosh et al. [10], who applied it only to transient faults. Such transient-fault containment was achieved by Demirbas et al. [8] for the problem of tracking in sensor networks. None of this research offers a broad approach to containing Byzantine faults.

Second, a number of researchers have investigated ways to limit Byzantine corruption when performing broadcast [3], multicast [22], or gossip [17, 20]. These closely related problems have no computational aspect; they merely propagate data. Furthermore, they have the property that correct operation implicitly replicates all of the data to all machines. The resulting redundancy enables machines to vote on the data’s correctness, as in the original Byzantine agreement problem.

Third, some researchers have tackled specialized subclasses of the general problem. Merideth [19] proposes a proactive fault-containment system that relies on fault detection, a well-known specialization of fault-tolerance problems [7]. Krings and McQueen [13] employ standard Byzantine-fault-tolerant protocols only for carefully defined “critical functionalities.” The TTP/C protocol [5] isolates only a constrained subset of Byzantine faults, namely reception failures and consistent transmission failures.

Thus, every known technique for building systems that resist Byzantine faults has at least one of the following weaknesses:

- Its throughput does not increase with scale.
- It addresses only a narrow academic problem.
- It does not support computation.
- It does not address general Byzantine faults.

3 CONTEXT – FARSITE FILE SYSTEM

We developed BFI in the context of a scalable, peer-to-peer file system called Farsite [2]. Farsite logically functions as a centralized file server, but it is physically distributed among the desktop workstations of a university or corporation, which may have over 10^5 machines. In this environment, independent Byzantine faults are significantly more likely than they would be in a physically secure server cluster.

Farsite employs different techniques for managing file content and metadata. File content is encrypted, replicated, and distributed among the machines in the system, and a secure hash of each file’s content is maintained with its metadata. File metadata is managed by BFT groups of workstations; we call each group a “server”. Every machine in a Farsite system fills three roles: a *client* for its local user, a *file host* storing encrypted content of data files, and a member of a BFT group that acts as a *server* for metadata.

File metadata is dynamically partitioned among servers, as follows: A Farsite system is initialized with a single server, called the *root*, which initially manages metadata for all files. When the metadata load on the root becomes excessive, it assembles a randomly chosen set of machines into a new BFT group, and it *delegates* a subset of its files to this newly formed server. This process continues as necessary, resulting in a tree of delegations. The fanout of the tree is a matter of policy.

The only difference between directories and data files is that the former may have no content and the latter may have no children. This is a small enough distinction that we refer to them both simply as “files”.

4 MOTIVATION

This section argues for the value of isolating Byzantine faults in a scalable peer-to-peer system. In particular, we consider a distributed file system, specifically a Farsite-like system of interacting BFT groups. For analytical simplicity, we assume that the system’s files are partitioned evenly among the constituent BFT groups, and we assume independent machine failure. For concreteness, we assume a machine fault probability of 0.001; in the analysis, we discuss our sensitivity to this value. We evaluate the *operational fault rate*, which is the probability that an operation on a randomly selected file exhibits a fault.

4.1 Model

If a third or more of the machines in a BFT group are faulty, the group cannot mask the fault. Therefore, if P_m is the probability that a machine is faulty, the probability that a group of size g manifests a fault is:

$$P_g = 1 - B\left(\lfloor (g-1)/3 \rfloor, g, P_m\right)$$

Function B is the cumulative binomial distribution function. In a system of n BFT groups, the probability that at least one group manifests a fault is:

$$P_s = 1 - B\left(0, n, P_g\right)$$

We consider three cases. In the first case, there is no fault-isolation mechanism, so a single faulty group may spread misinformation to other groups and thereby corrupt the entire system. The probability that any given file operation exhibits a fault is thus equal to the probability P_s that the system contains even one faulty group. This is shown by the three dashed lines in Fig. 1 for BFT groups of size 4, 7, and 10, as the count of groups scales up to 10^5.

The second case illustrates ideal fault isolation. Each BFT group can corrupt only operations on the files it manages, so the probability of a faulty file operation equals the probability P_g that the file's managing group is faulty. System scale is thus irrelevant, as illustrated by the dark line in Fig. 1 for 4-member BFT groups.

The third case illustrates BFI in Farsite. Pathname lookups involve metadata from all files along the path, so a faulty group can corrupt lookups to its descendent groups' files. Recall that the count of nodes in a tree with l levels and node fanout of f is:

$$N(f,l) = \left(f^l - 1\right) / \left(f - 1\right)$$

In a tree of $N(f, l)$ BFT groups, the expected number of groups that are faulty or have a faulty ancestor is defined by the recurrence:

$$F(f,l) = P_g \cdot N(f,l) + \left(1 - P_g\right) \cdot f \cdot F(f, l-1)$$

$$F(f,1) = P_g$$

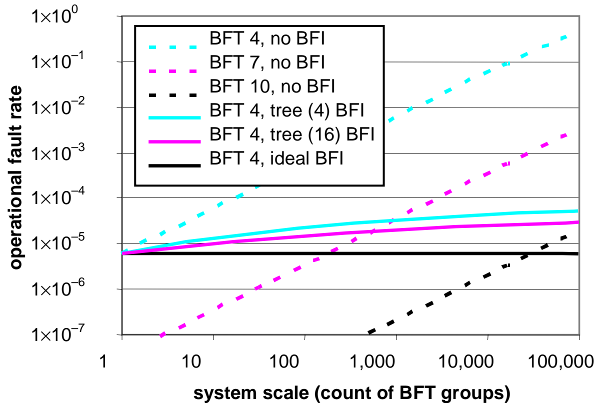

Fig. 1: Faults at scale (machine fault probability = 0.001)

Thus, the probability of a faulty absolute-path-based file operation in a system of $N(f, l)$ BFT groups is:

$$P_t = F(f,l) / N(f,l)$$

This is illustrated by the light and medium lines in Fig. 1 for systems in which each 4-member BFT group has 4 or 16 children, respectively. Since not all file operations are path-based, and since paths can be specified relative to open files, the operational fault rate will actually be somewhat lower than that indicated by Fig. 1.

4.2 Analysis

As the light dashed line shows, a single 4-member group has an operational fault rate of 6×10^{-6}, which is better than five nines. However, when the scale reaches 10^5 groups, the operational fault rate is 0.45; almost half of all operations exhibit faults. By contrast, with a group fanout of 16, a 10^5-group Farsite system exhibits an operational fault rate of 3×10^{-5}, showing that fault tolerance is largely preserved as scale increases.

An alternate way to improve the large-scale operational fault rate is to increase the size of BFT groups. However, as Fig. 1 shows, in a system of 10^5 groups, the group size must be increased from 4 to 10 to achieve the operational fault rate that BFI achieves. This increase cuts the system throughput by 60% (= 1 – 4/10) or more, because it increases the redundant computation by a factor of 2.5 (= 10 / 4) and the intra-group message traffic by a factor of 6.25 (= $10^2 / 4^2$).

The curves in Fig. 1 are based on a machine fault rate of 10^{-3}. For higher rates, the benefit of BFI is even greater, since a larger increase in group size (and corresponding drop in throughput) is needed to achieve the same operational fault rate as BFI. By contrast, a lower machine fault rate reduces the benefit, although not by much: Even with a machine fault probability of 10^{-5} in a system of 10^5 groups, BFI enables 4-member groups to achieve an operational fault rate that would otherwise require 7-member groups and an attendant drop in throughput of over 42% (= 1 – 4/7).

5 BYZANTINE FAULT ISOLATION

BFI is a technique that prevents a faulty BFT group from corrupting an entire system. The technique is based on using formal specification to design a distributed system with well-defined semantics [15]. BFI semantically specifies the faulty behavior that can manifest when faults occur in the distributed system.

Several approaches may be used to validate that the system design adheres to the specified fault semantics. In our work, we used only informal proof, but greater confidence can be attained through model checking [15] or mechanically verified formal proof [9]. Our limited experience with such methods suggests that they would be challenging to apply to a formal spec as big as ours.

5.1 Formal Distributed System Specification

We follow an approach to formal specification advocated by Lamport [15]. This approach has three components: a semantic spec, a distributed-system spec, and a refinement.* Each of the two specs defines *state* information and *actions* that affect the state.

A *semantic specification* describes how users experience the system. Farsite logically functions as a centralized file server, so Farsite's semantic spec defines the behavior of a centralized file server. The state defined by the semantic spec includes files, handles, and pending operations. The actions defined by the semantic spec are file-system operations: open, close, read, write, create, delete, and move/rename. For example, the semantic spec says that the open operation resolves a pathname, checks access rights to the file, adds a new handle to the set of handles open on the file, and returns the handle to the caller. The spec also describes the error codes returned if a pathname fails to resolve or the access rights are invalid. In sum, the semantic spec characterizes the system from the users' point of view.

A *distributed-system specification* describes how a set of machines collaborate to produce a desired behavior. The state defined by the distributed-system spec includes machines, abstract data structures maintained by machines, and messages exchanged between machines. The actions defined by the distributed-system spec are tasks performed by individual machines or BFT groups: sending messages, receiving messages, and modifying local-machine state. For example, the distributed-system spec says that when a server receives a message from a client asking for the necessary resources to perform an open operation, if the server has the resources available, it sends them to the client; otherwise, it begins the process of recalling the resources from other clients that are holding the resources. In sum, the distributed-system spec characterizes the system from the system designers' point of view.

A *refinement* is a formal correspondence between the semantic spec and the distributed-system spec. It describes how to interpret distributed-system state as semantic state and distributed-system actions as semantic actions. For a state example, the refinement may say that the attributes of a particular semantic-level file are defined by a particular data structure on a particular machine in the distributed system; the particulars express the system designer's rules for what data means and which data is authoritative.

Actions in the distributed-system spec are classified as either foreground or background actions, according to whether they have semantic effects. A foreground action corresponds to an action in the semantic spec; for example, updating a certain distributed-system data structure may correspond to the completion of an open operation in the semantic spec. A background action corresponds to a non-action in the semantic spec; for example, passing resources to a client by means of a message has no semantic effect.

The basic distributed-system spec defines interactions among a set of non-faulty machines. The basic semantic spec defines the user-visible behavior of a fault-free system. If the system design is correct, the refinement will show that the distributed-system spec satisfies the semantic spec.

5.2 Specifying Failure Semantics

To understand the system's behavior under fault conditions, we modify the distributed-system spec as follows: We augment the state of each machine with a new flag indicating whether the machine is *corrupt*, and we add a new action that sets a machine's corrupt flag. When a machine is corrupt, its local state is undefined, and it can send arbitrary messages to other machines at any time.

These modifications to the distributed-system spec prevent it from refining to the basic semantic spec. The BFI technique proceeds by progressively modifying the two specs until the distributed-system spec again satisfies the semantic spec:

- The semantic spec is weakened to describe how faults appear to the users of the system.
- The distributed-system spec is strengthened to quarantine faults from non-corrupt machines.

The art of Byzantine fault isolation is in determining what modifications to make to the two specs. An overly weak semantic spec may not isolate faults sufficiently; an overly strong distributed-system spec may not facilitate a practical and performant implementation; and a semantic spec that is too strong or a distributed-system spec that is too weak will not admit a satisfactory refinement.

In Farsite, we modified the semantic spec by associating a flag with each file to indicate whether the file is *tainted*, and we added an action that sets the flag. We established a refinement in which a file becomes tainted if and only if the BFT group that manages the file becomes corrupt.

It would have been ideal to weaken the semantic spec so little that tainted files could not corrupt operations on non-tainted files. Unfortunately, because path-based file operations involve metadata from all files along the path, we were unable to design a distributed system that refined to such a semantic spec. We thus had to weaken the semantic spec further, permitting tainted files to lie about their parents and children and thereby to corrupt path-based operations on the tainted file's descendents.

---

* Lamport uses different terminology because his approach is applicable to a broader class of problems than distributed systems.

We were, however, able to constrain those lies to prevent a tainted file from affecting operations on files elsewhere in the namespace. In particular, a tainted file cannot falsely claim an existing non-tainted file as its child or parent. Exhaustively, the weakened safety semantics allow a tainted to appear to…

(1) …have arbitrary contents and attributes,
(2) …not be linked into the file tree,
(3) …not have children that it actually has,
(4) …have children that do not actually exist, or
(5) …have another tainted file as a child or parent.

In addition, the weakened liveness semantics allow operations involving a tainted file to not complete.

The modifications to Farsite's distributed-system spec are far more involved. Some of the key principles include maintaining redundant information across BFT group boundaries, augmenting messages with information that justifies their correctness, ensuring unambiguous chains of authority over data, and carefully ordering messages and state updates for operations involving more than two BFT groups. We illustrate an example in the next section.

6 BFI EXAMPLE: MOVE OPERATION

Our BFI modifications to the distributed-system spec are too extensive to permit even a high-level summary in this paper. Nonetheless, to convey the flavor of these modifications, we outline one example of how we modified Farsite's distributed-system spec to satisfy the semantic spec. The example exploits a redundancy that we added for BFI, namely that parent-child links are stored by both parent and child files. If a client observes inconsistent information about the link between a parent and a child, the client regards that link as nonexistent.

The specific example we present is a component of the most complex file-system operation, *move*, whose semantics are that an *object* file has its parent changed from a *source* file to a *destination* file, thereby changing the full pathnames of all descendents of the object file.

The object, source, and destination files might be managed by three separate servers, each of which may be faulty or not. We will not present our full case analysis here, but we will describe the highlights.

If all non-faulty servers agree on whether the move succeeds, the refinement defines the semantic result as the result obtained by the non-faulty servers. Detailed analysis shows that this satisfies the fault semantics enumerated above. This rule also covers the case in which all servers are faulty, because then any semantic result is consistent with the fault semantics.

Case		Object	Source	Dest.	Semantic
1	*a*	**faulty**	success	failure	either
	b	**faulty**	failure	success	none
2	*a*	success	**faulty**	failure	failure
	b	failure	**faulty**	success	none
3	*a*	failure	success	**faulty**	success
	b	success	failure	**faulty**	none

Table 1: Fault case analysis for move operation

If no servers are faulty, our protocol ensures that all servers agree on the result.

The tricky cases are those in which one server is faulty and the other two disagree on the result. It would be ideal to somehow prevent these cases from ever occurring, but this is provably impossible [23]. As Table 1 shows, for each case (faulty object, faulty source, and faulty destination), refinement can select a satisfactory semantic result if the other servers disagree in a particular way (the *a* subcases) but not if they disagree in the opposite way (the *b* subcases).

For example, in case 1, the object server is faulty. In subcase *a*, the source believes the move to be successful so it unlinks the object, but the destination believes the move to be unsuccessful so it fails to link the object. Consequently, the object becomes unlinked from the file tree. However, safety weakness 2 (see previous section) states that a tainted file may appear to not be linked into the file tree. Thus, our refinement could interpret the outcome either as a tainted file successfully moving and then not appearing linked into the tree or as a tainted file failing to move and then not appearing linked into the tree.

The analysis of the *a* subcases for cases 2 and 3 is similar albeit slightly more complex, because 2*a* must be interpreted as a failed move and 3*a* must be interpreted as a successful move, to be consistent with the safety weaknesses allowed by the failure semantics.

In the *b* subcases, no semantic result is consistent with the distributed-system outcome. For example, in subcase 1*b*, the destination links the object but the source does not unlink it. If the refinement were to interpret this as a successful move, the destination file would become the object's parent, but because the source thinks the move failed, it still believes that it is the object's parent, so the object could pretend that the source is in fact its parent, which our failure semantics disallow. A similar argument holds for interpreting the result as a failed move. Since it is impossible to refine the subcase *b* outcomes, Farsite must prevent them.

Our protocol precludes the *b* subcases by ensuring that the non-faulty servers observe certain ordering constraints before declaring a move operation successful: The object server does not commit until after the source server commits, and the destination server does not commit until after the source and object servers commit. Examination of the table shows that this prevents the problematic subcases. For example, subcase 1*b* cannot occur because the destination will not declare success until it first hears that the source has declared success, which it has not.

7 SUMMARY

Although Byzantine Fault Tolerance (BFT) allows a trusted service to run on a peer-to-peer system of untrusted machines, it does not support scaling up to increase system throughput. Scale-up can be achieved by partitioning a workload among multiple BFT groups, but this leads to an increase in the probability of total system failure as the system scale increases.

To solve this problem, we introduced Byzantine Fault Isolation (BFI), a methodology for using formal specification to constrain the semantic behavior of a faulty system. BFI modifies a system design to formally recognize that machines may become corrupt, wherein they have undefined local state and can send arbitrary messages to other machines. These modifications highlight areas that require design changes to restrict the spread of faulty information.

Even in the presence of design features that restrict faults, corrupt machines may still affect the system's semantics. Thus, the BFI technique involves modifying the system's defined semantics to reflect the partial correctness achievable when the system is degraded. BFI uses formal specification to ensure that the modified system design satisfies the modified system semantics.

We quantified the benefit of BFI to scalable systems: In a tree-structured system of 10^5 BFT groups, wherein a faulty group can corrupt its descendents' operations, BFI can enable 4-member BFT groups to achieve the same operational fault rate as 10-member BFT groups, without the corresponding 60% drop in throughput due to increased replication and message traffic.

We employed the BFI technique in the design of the Farsite distributed file system, a large and complex peer-to-peer system designed to scale to 10^5 machines. BFI guided us toward adding specific redundancies, enriching messages, restricting authority, and constraining the order of distributed operation steps. Using BFI, we related these design changes to the system's semantics, thereby showing that file corruption cannot spread to unrelated regions of the file-system namespace.

Prior to BFI, no technique has addressed how to interconnect multiple BFT groups in a way that isolates Byzantine faults.

REFERENCES

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, J. Wylie. "Fault-scalable Byzantine Fault-Tolerant Services." 20th SOSP, 2005.

[2] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, R. P. Wattenhofer. "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment." 5th OSDI, 2002.

[3] Y. Azar, S. Kutten, B. Patt-Shamir. "Distributed Error Confinement." PODC, 2003.

[4] M. Barborak, M. Malek. "The Consensus Problem in Fault-Tolerant Computing." ACM Computing Surveys, 25(2), 1993.

[5] G. Bauer, H. Kopetz, W. Steiner. "Byzantine Fault Containment in TTP/C." Workshop on Real-Time LANs in the Internet Age, 2002.

[6] M. Castro, B. Liskov, "Practical Byzantine Fault Tolerance." 3rd OSDI, 1999.

[7] T. D. Chandra, S. Toueg. "Unreliable Failure Detectors for Asynchronous Systems." JACM 43(2), 1996.

[8] M. Demirbas, A. Arora, T. Nolte, N. Lynch. "A Hierarchy-Based Fault-Local Stabilizing Algorithm for Tracking in Sensor Networks." 8th OPODIS, 2004.

[9] U. Engberg, P. Grønning, L. Lamport. "Mechanical Verification of Concurrent Systems with TLA." 4th Computer Aided Verification, 1992.

[10] S. Ghosh, A. Gupta, T. Herman, S. V. Pemmaraju. "Fault-Containing Self-Stabilizing Algorithms." PODC, 1996.

[11] M. P. Herlihy, J. M. Wing. "Specifying Graceful Degradation." IEEE TPDS 2(1), 1991.

[12] K. P. Kihlstrom, L. E. Moser, P. M. Melliar-Smith. "The SecureRing Protocols for Securing Group Communication." Hawaii International Conference on System Sciences, 1998.

[13] A. W. Krings, M. A. McQueen. "A Byzantine Resilient Approach to Network Security." 29th International Symposium on Fault-Tolerant Computing, 1999.

[14] L. Lamport, R. Shostak, M. Pease. "The Byzantine Generals Problem." ACM TPLS, 4(3), 1982.

[15] L. Lamport. *Specifying Systems*. Addison-Wesley, 2003.

[16] C.S. Lewis, J. Saia. "Scalable Byzantine Agreement." Technical Report, University of New Mexico, 2004.

[17] D. Malkhi, Y. Mansour, M. K. Reiter, "On Diffusing Updates in a Byzantine Environment." 18th SRDS, 1999.

[18] T. Masuzawa, S. Tixeuil. "A Self-Stabilizing Link-Coloring Protocol Resilient to Unbounded Byzantine Faults in Arbitrary Networks." Laboratoire de Recherche en Informatique Report #1396, 2005.

[19] M. G. Merideth. "Enhancing Survivability with Proactive Fault-Containment." DSN Student Forum, 2003.

[20] Y. M. Minsky, F. B. Schneider. "Tolerating Malicious Gossip." Distributed Computing, 16(1), 2003.

[21] M. Nesterenko, A. Arora. "Tolerance to Unbounded Byzantine Faults." SRDS, 2002.

[22] V. Pappas, B. Zhang, A. Terzis, L. Zhang, "Fault-Tolerant Data Delivery for Multicast Overlay Networks." ICDCS, 2004.

[23] M. Pease, R. Shostak, L. Lamport. "Reaching Agreement in the Presence of Faults." JACM 27(2), 1980.

[24] M. K. Reiter. "The Rampart Toolkit for Building High-Integrity Services." TPDS (LNCS 938), 1995.

[25] Y. Sakurai, F Ooshita, T. Masuzawa. "A Self-stabilizing Link-Coloring Protocol Resilient to Byzantine Faults in Tree Networks." 8th OPODIS, 2004.

[26] F. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial." ACM Computing Surveys, 22(4), 1990.