Deferring Trust in Fluid Replication

Brian D. Noble, Ben Fleis, and Landon P. Cox
University of Michigan 1301 Beal Avenue
{bnoble,lbf,lpcox}@umich.edu Ann Arbor, MI, 48109-2122

Abstract

Mobile nodes rely on external services to provide safety, sharing, and additional resources. Unfortunately, as mobile nodes move through the networking infrastructure, the costs of accessing servers change. Fluid replication allows mobile clients to create replicas where and when they are needed. Unfortunately, one must trust the nodes holding these replicas, and establishing trust in autonomously administered nodes is a difficult task. Instead, we argue that trust should be *deferred*. In this position paper, we present the design of *Stonewall*, a system that defers trust decisions through the use of two mechanisms: *packages* and *receipts*. The former ensure confidentiality and detect breaches of integrity; the latter detect breaches of non-repudiation.

1 Introduction

Hand-held, mobile nodes are becoming indispensable members of the computing infrastructure. Unfortunately, such clients suffer from several intrinsic challenges that have rendered them second-class citizens. These include variable networking performance, scarcity of local resources, and increased susceptibility to theft, loss, and destruction. Such devices are often too small to retain all useful files, and storing data solely on them increases the risk of loss; instead, they must rely on server support. However, as the user moves, the costs of interacting with home servers change. This gives rise to bouts of poor performance and decreases the utility of servers.

One way to solve this is through *fluid replication* [13], the automatic creation of replicas where and when they are needed. In fluid replication, clients monitor the networking performance between themselves and their servers. When network performance to a server becomes poor, clients instantiate a replica of that server on a nearby node called a *WayStation*. Thereafter, the client interacts with a WayStation over a high-quality network path. The WayStation and server maintain consistency of updates between themselves through a periodic process called *reconciliation* [8, 14]. In this way, fluid replication can provide the benefits of server support while deferring or eliminating most traffic across the bottleneck path.

For fluid replication to be useful, there must be many WayStations conveniently located throughout the networking infrastructure. Together, they act as a loose confederation of nodes providing service to users as they move through the network. However, it is important that WayStations remain under local, autonomous control; their creation, placement, and administration must be completely decentralized to preserve administrative scalability.

This gives rise to some interesting trust and security concerns. When a client interacts with a WayStation, it must be assured of three things. The first is *confidentiality*: data should not observed or exposed by the WayStation while it is stored there. The second is *integrity*: WayStations or other untrusted parties should not modify stored data. Third, WayStations must provide *non-repudiation* of accepted updates; they must be propagated to the server for which they are destined. Together, these three properties form a *contractual relationship* between WayStations and clients; the latter must trust that the contract will be *performed* before committing to the relationship.

This is an instance of the trust management problem [2]. There are several ways one might establish trust, but none of them are ideal. One could require all parties to pre-establish trust relationships, but this will not scale due to the number and dynamic nature of administrative interests. One could depend on schemes such as paths or webs of trust [3] to establish trust as it is needed. Unfortunately, as the path of trust establishment lengthens, the overall path suffers from *trust dilution*; solving this problem often requires interactions with many remote sites [15].

Rather than require that clients trust WayStations before using them, we argue instead for *deferring the need for trust*. We propose to do so through *Stonewall*, a system that renders breaches of confidentiality impossible, and ensures that breaches of integrity or non-repudiation are detectable. Together, these properties enable verification of contract performance, removing the need to establish trust from the critical path of replica instantiation.

Stonewall relies on two abstractions: *packages* and *receipts*. Each package encrypts a file block and a hash of that block. When given an updated package, a WayStation responds with a receipt, comprising a signed hash of the update. Encrypting packages renders breaches of confidentiality impossible. The inclusion of hashes and generation of receipts gives clients a way to detect breaches of integrity and non-repudiation.

The simple approach of issuing receipts for every update raises two concerns. First, the public-key techniques necessary for receipts are computationally expensive. Second, requiring a receipt for each update would require clients to store redundant receipts and WayStations to forward redundant updates. We solve the first problem by batching receipts at the granularity of reconciliation, and the second through receipt cancellation among clients of a WayStation.

2 Background: Fluid Replication

As mobile devices move through the networking infrastructure, the costs to access their home services change. One approach to solving this problem is peer-to-peer replication [5, 8, 14]. This addresses the problem of variable performance, but introduces its own difficulties. First, since update propagation is dependent on client mobility and communication patterns, one cannot offer bounds on the rate of update convergence. Second, client machines — particularly mobile clients — are inherently less trustworthy than servers [7]; they are more easily lost, stolen, or compromised. Third, the smallest mobile devices are not capable of long periods of autonomous operation. They cannot cache enough data for long-term use, nor can they retain all updates. Instead, they must periodically interact with servers.

Fluid replication, the automatic creation of replicas when and where they are needed, specifically addresses the performance problems of mobile clients without introducing the difficulties of peer-to-peer replication. In fluid replication, clients are supported by a loose confederation of infrastructure nodes, called WayStations. Each client monitors its performance in accessing a remote server. If performance becomes unacceptable, a replica of that server is created on a WayStation near the client. WayStations are administered locally, but can offer file replication services both to local and visiting clients. These might be offered on a pay-as-you go basis, or through a cooperative agreement.

The key to good WayStation performance is the careful balancing of consistency with the costs of communicating over wide-area networks. In order to hide these communication costs, WayStations and servers rely on *optimistic* consistency control whenever possible. In optimistic schemes, each replica site logs updates, and periodically reconciles these updates with those performed at the replicated server. The reconciliation interval is based on update rate and pattern, network performance, and application advice.

From the point of view of Stonewall, there are three facets of fluid replication that are particularly significant: replica population, reconciliation, and log optimization. Each of these is described below. Other details of our fluid replication design — network performance monitoring, distance-based discovery, and replica migration — are presented elsewhere [13]. The features described here are similar to those provided by Coda [8, 11], but differ in some details.

2.1 Replica Population

Once a client decides it can profitably benefit from a replica on a particular WayStation, the client asks it to instantiate that replica. This involves no data copying; the replica is populated lazily. Logically, each WayStation forms a two-way replica with the home server; the server manages multi-site, dynamic replication.

After establishing a replica on a WayStation, the client directs all of its read and write requests there. These requests are block-oriented; they name a file, a block within the file, and (for writes) supply the new contents of the block. If a read request arrives that cannot be directly satisfied, the WayStation fetches the relevant block from the remote server on demand.

Writes are sent from the client to the WayStation as normal, but are not reflected back to the home server immediately. Instead, writes are collected at WayStations rather than immediately reflected to the home server. These writes, stamped with logical clock time [9], form a virtual log at the WayStation. Periodically, these logs are exchanged in the reconciliation process.

2.2 Reconciliation

When a reconciliation begins, the server first checks to see which updates in the log are *serializable* with updates that have already been performed at the server. Each block contains a version stamp based on logical clock time; write operations are checked to make sure they do not conflict. Updates made only at the WayStation are applied at the server. Blocks updated only at the server are marked invalid at the WayStation. If a block of a file was written both at the WayStation and the server, the file containing the block is marked in *conflict*, preventing further use of the inconsistent file. Conflicts must be manually resolved before the file can be used again. This approach provides session consistency, where a session is defined as the reconciliation interval.

2.3 Log Optimization

A central issue in providing optimistic replication is growth of update logs. Each node must retain all log entries that mention updates that another replica has not yet seen. However, there are two classes of optimizations that can be made; they both depend on the notion of a replica interval.

The first class of log optimizations is the elimination of redundant or self-canceling sets of operations within a replica interval. Redundant operations include updates to the same object; only the last update need be maintained. Self-canceling operations are those that, when composed, do not change persistent state.

The second class of optimization is the truncation of a log's stable prefix — the log entries that are known to be unnecessary from now on. For a WayStation, the stable prefix of the log is the portion before the last interaction with the remote server. For the server, the stable prefix is defined as the portion of the log known to all replicas.

3 Design Considerations

There are two considerations that have driven our design of Stonewall. The first is our assumed trust and threat model. The second arises from the interaction between concerns of scalability and security.

3.1 Trust and Threat Model

Fluid replication is a mechanism augmenting existing file services. Therefore, we assume that servers have a pre-existing way of identifying clients and authorizing their reads or writes to individual files. Further, we assume that clients have some way of determining the identity of servers and imbuing those servers with the trust to accept updates made by clients. Thus, Stonewall does not concern itself with these mechanisms.

In contrast, WayStations can be administered and advertised by arbitrary authorities. They need not validate the identities of clients requesting replica services, and might attempt to observe, divulge, or change any replica contents that they store. Further, they may choose not to retain update logs or file contents when space becomes scarce. We also do not assume that the network provides any security services to higher layers.

However, Stonewall does assume that principals — clients, servers, and WayStations — hold a public/private key pair, and that all principals can obtain the public key of any other principal in the system. Given our trust model this is easy to require of the set of clients and servers. For WayStations, we depend on the presence of a public key infrastructure [4, 6] or alternate key management system [10]. Note that merely knowing the public key of a WayStation does not automatically convey a trust relationship; it merely establishes the identity of that WayStation.

3.2 Scalability and Security

One of the central premises of Stonewall is the possibility of removing WayStations from the path of trust establishment. This means that the responsibility for providing trust mechanisms must reside with either the server or its clients. Because servers are likely to be the bottleneck to scalability [7], we have chosen to place computational burdens with the clients. While this is likely to increase the scalability of the overall system, it is unclear whether mobile clients — which are resource-poor — can support this demand. There is a tradeoff between the collective computational power of clients, and the demands on central servers. If the burden on clients is unreasonable, one can instead imagine using proof-carrying code [12] or similar techniques to install code safely on the WayStation to perform some of these operations.

4 Stonewall Design

Fluid replication depends on the presence of WayStations throughout the infrastructure. These WayStations must be administered autonomously; to scale, one cannot require a priori establishment of trust relationships. Instead, Stonewall defers trust establishment, placing the burden on clients and servers to provide *confidentiality*, *integrity* and *non-repudiation*.

Confidentiality and integrity are provided by the fundamental unit of transfer, the *package*, described in Section 4.1. *Receipts* are counterpart to packages. They provide a way to detect breaches of non-repudiation, and are presented in Section 4.2. Implementing receipts in the naive way incurs unreasonable computational and data storage/transfer overheads. These are addressed in Sections 4.3 and 4.4.

$x + y$	$\equiv$	concatenate x, y
$H(x)$	$\equiv$	hash of x
K^c	$\equiv$	public key for client c
$K_{i,j}$	$\equiv$	session key for file i on WayStation j
$e_K(x)$	$\equiv$	encrypt x using key K
$d_K(x)$	$\equiv$	decrypt x using key K
$s_K(x)$	$\equiv$	sign x using key K

Figure 1: **Stonewall Notation**

4.1 Packages

The package is the Stonewall abstraction of a data block. It is defined as:

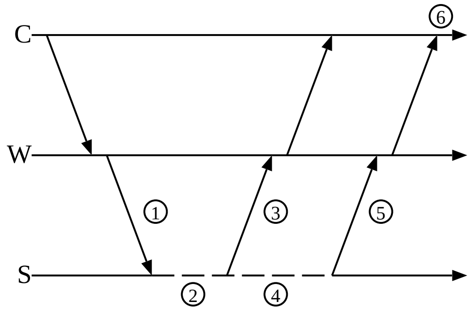
$$P_b = P(K, b, id) = e_K\Big(b + id + H(b + id)\Big).$$

where b is the data block, id is a versioning identifier, and H is a hash of these. Identifiers include the name of the host that generated the block and a logical timestamp that serves as both a version number and a nonce. The hash provides a means to verify integrity; encrypting the contents provides confidentiality. The notation used in this and other definitions is shown in Figure 1.

The symmetric key used in encrypting packages is assigned on a per-file, per-WayStation basis, and is short-lived. This session key is shared strictly between the home server and the clients authorized to access the file, and thus acts as a capability. Assigning keys in a fine-grained way reduces the impact of compromised keys, and enables re-keying of files by clients when necessary.

These keys are distributed from servers to clients in response to replica population at a WayStation. Each session key is sealed with the public key of the requesting client, providing assurance that only authorized clients can make use of them.

Once a client has a package and the associated key container, the file key is extracted from its container and retained. The session key is used for all accesses to the fetched object. The package is decrypted with the session key, and the hash value is checked. If the hash matches the stored value, the client has verified integrity of the package contents. The exchange of file contents and session keys is illustrated in Figure 2. There are several opportunities to take advantage of prefetching to reduce or eliminate client-perceived latency in obtaining session keys. For example, servers can pre-encode session keys and packages based on known access patterns. The session keys can be batched to amortize their costs, and packages can be forwarded asynchronously.



(1) W requests P_b, and K from S, (2) S generates a random K, and creates P_b, (3) S sends P_b to W, (4) S creates $e_{K^C}(K, id)$ where e_{K^C} is decryptable by C, (5) S sends $e_{K^C}(K, id)$ to W, and (6) C can now use block b .

Figure 2: **Fetch Action Sequence: obtaining P_b**

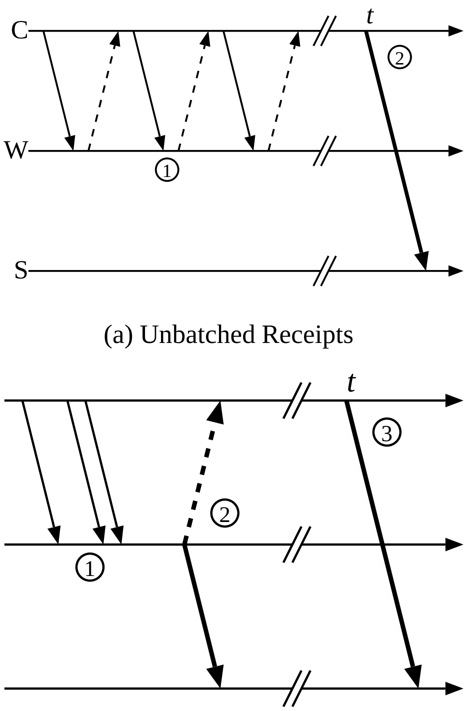
Logically, keys are selected by the server, and therefore the server is responsible for re-keying files when old keys expire. There are two unfortunate implications of this. First, the slow path between WayStation and server contributes to the time needed for re-keying, and likely dominates. Second, it reduces the scalability of the system by increasing server CPU load. Instead, we take advantage of the key distribution mechanism to divide the re-keying work amongst the clients of a WayStation. How best to balance the costs of re-keying at resource-poor mobile clients against the gains in scalability and network performance is an open question.

4.2 Receipts

Packages prevent breaches of confidentiality, and render breaches of integrity detectable. However, they do not provide non-repudiation of accepted updates. Packages sent to the WayStation may be transparently dropped there. Since clients only affect each others' logical clocks through interactions with the WayStation, they could not be used to detect dropped updates. Clients could log updates and check whether they were propagated, but would not be able to prove that the WayStation had taken responsibility for those not delivered.

There are many reasons why a WayStation might drop updates. For example, it may be short of storage space, and decide to flush any updates from foreign clients. Stonewall can not prevent loss of data, because WayStations act with complete autonomy. Instead Stonewall focuses on provably detecting these failures. It does so through the use of receipts. When sending a package via overnight courier, the courier gives the sender a receipt. This acts as proof that the courier has accepted responsibility for the package. Combined with the absence of a delivered package, it can be used to demonstrate that a courier failed to deliver as promised. Likewise, WayStations issue receipts for updated packages that they accept.

A receipt R is a digitally signed copy of a hash of P_b and a current id; it is signed with the WayStation's private key.



(a) Unbatched Receipts

(b) Batched Receipts

In Figure 3(a), (1) W generates a receipt each time it receives a client update. (2) At time t, C submits all of its receipts directly to S for confirmation.

In Figure 3(b), (1) W accepts updates without immediately generating a receipt. (2) During reconciliation, W gives a batch receipt to C for all unreconciled updates. (3) At time t, C submits all of its receipts directly to S for confirmation.

Figure 3: **Receipt Generation**

More formally a receipt is

$$R = s_K\Big(H(P_b) + id\Big).$$

R acts as a signed acknowledgement. The current id is included so that package submission and acceptance become distinct logical events; the id also acts as a nonce which prevents the WayStation from having to give out blind signatures. A hash value is calculated for the submitted package, and recorded in the receipt. This hash provides a compact handle for referencing the received package.

After submitting an update to a WayStation, the client must retain the updated package until it receives a receipt for it. Only then can the client be assured that any failure to propagate the update can be detected. As clients submit updates, they must retain receipts until they can be confirmed at the server. Typically, confirmation is deferred until the client and server are strongly connected. This process is illustrated in Figure 3(a).

When the client asks the server to confirm receipts, the server performs three steps. First it must verify the signature on the receipt. Second it examines its own logs to find the next reconciliation occurring after the id found in the receipt. Once the reconciliation is found, the third step is a comparison of the signed hash value with the hash of the package received during this reconciliation. If the signature or hash comparison fail, the receipt is considered bogus, and the WayStation has failed to uphold its contract. If all steps succeed, the server knows that the update named in the receipt was propagated.

4.3 Managing Computational Overheads

Although this simple system is functionally correct, its performance is likely to be unacceptable. Generating an asymmetric key signature at every client update would be nearly impossible for even a moderately busy WayStation. However, client-server receipt confirmations are likely to be infrequent. They are typically performed only when the client and server are well-connected, at which point the client would no longer make use of WayStation services. Furthermore, receipt confirmation cannot possibly happen before the reconciliation during which the corresponding update was to have propagated.

In light of this, one can easily *batch* receipts, and issue them in bulk at reconciliation time. This amortizes the expense of computing the signature across potentially many updates. We chose to issue receipts at reconciliation time for a number of reasons. We do not wait any longer because the WayStation is already perusing and clearing its update logs; delaying receipts only forces the WayStation to retain state longer than necessary. We do not issue receipts more quickly because there is no way to confirm until the reconciliation interval has passed. Therefore, Stonewall uses receipt bundling to offer identical safety to that provided by immediate issuance, while amortizing expensive signatures.

The main drawback to bundling is the increased pressure on client resources. Mobile clients — which by their very nature are resource-poor — must retain copies of packages until they are issued receipts. Therefore, the client must be able to request receipts on demand. While they cannot be confirmed until the next reconciliation, they do allow the client to reclaim potentially scarce resources. The ability to generate receipts on demand does not place additional burdens on the design of fluid replication; WayStations must already provide reconciliation on demand in order to support session semantics [16] to clients that migrate to another location.

Clients typically confirm receipts only when the connections between themselves and their servers are strong. However, there are two reasons why a client might choose to confirm receipts before it is well connected. First, the client may need to reclaim space. While receipts are much smaller than packages, the former tend to be longer-lived. Mobile clients that are far away from servers for some time can find this overly burdensome. Second, a client may wish to discover lost updates within a time bound; such clients may not have the luxury of waiting for strong network connectivity. We plan to investigate this tradeoff between performance and bounded safety.

4.4 Managing Data Overheads

Ordinarily, locality in update access patterns [1] leads to several optimizations. WayStations make use of this locality to provide the log optimizations described in Section 2.3. These optimizations manage the growth of storage required at WayStations and, more importantly, reduce the amount of traffic required between a WayStation and a server.

Unfortunately, issuing a receipt for each update renders the benefits of log cancellation useless. In order for a client to confirm each receipt with a server, each individual update must be propagated. The server and client cannot trust the WayStation to properly account for log cancellation, since that would imply that the WayStation was trusted. For example, consider two clients using a WayStation to share files with good performance. The first client creates a temporary file, and the second consumes and deletes it, all within a single reconciliation interval. The deletion should lead to cancellation of the transient file, but the first client will expect to be issued a confirmable receipt.

However, while clients cannot trust the WayStation, they can form trust associations with each other based on the rights each one has to a file. The server performs this trust judgment, since these rights are encapsulated by the granting of a session key to the client. This is as it would be without fluid replication. We leverage this within a reconciliation interval in the following way. When a client obtains a file block from the server, it must be informed of any pending receipts to be generated for that block. If it updates the block, it also generates a cancellation token for that receipt, signed by the session key for that file. The client expecting the first receipt will instead receive the cancellation token, which must have been generated by a client authorized to perform the canceling action.

If a client must specify all cancellations and log reductions, a WayStation can easily guess most of the operations that occur over the canceled event set. This optimization then implies that file operations may lose their opacity to WayStations. When a client accepts a cancellation, it reveals information about its usage patterns, which contrasts directly with the goal of operation confidentiality. Unfortunately, we believe this is unavoidable without resorting to token generation for every update sent to a WayStation; doing so seems to be a large cost for such a small benefit.

Cancellation tokens are only generated by clients within a reconciliation interval. Across such intervals, cancellation tokens are not generated or passed between servers and WayStations. This means that clients may have to store some extra receipts, but does not lead to any extra traffic. On the contrary, passing such cancellation tokens across reconciliations only increases the traffic along the expensive WayStation/server path. The server need not actually store the redundant data blocks; retaining receipt information is enough to later confirm client receipts. We believe this tradeoff is acceptable, since receipts are small compared to the blocks they cover.

5 Conclusion

Mobile clients experience wide variations in performance when accessing their home services from different locations. Fluid replication promises to address the sources of these performance problems by creating replicas on nearby WayStations — nodes in the infrastructure that provide replication services. For scalability, WayStations must be managed by their local administrative authorities, raising significant questions of security and trust.

Forcing clients to establish this trust before using a WayStation is an expensive proposition. Instead we propose to defer judgments of trust until they are actually needed. Our architecture for providing this, Stonewall, relies on two abstractions: packages and receipts. Together, these render breaches of confidentiality impossible, and make breaches of integrity and non-repudiation detectable. However, implementing these abstractions requires some thought and care to minimize the storage, processing, and networking costs. This work, when completed, will result in a global, distributed system that provides for vastly improved performance for mobile clients without a concomitant reduction in safety or security.

Acknowledgements

This research was supported in part by Novell, Inc.; the National Science Foundation under grant CCR-9984078; and the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0508. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Novell, Inc., the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, the National Science Foundation, or the U.S. Government.

References

[1] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 198–212, Pacific Grove, CA, USA, October 1991.

[2] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings 1996 IEEE Symposium on Security and Privacy.*, pages 164–73, May 1996.

[3] D. W. Chadwick, A. J. Young, and N. K. Cicovic. Merging and extending the PGP and PEM trust models — the ICE-TEL trust model. *IEEE Network*, 11(3):16–24, May–June 1997.

[4] W. Ford. Advances in public-key certificate standards. *SIGSAC Review*, 13(3):9–15, July 1995.

[5] J. S. Heidemann, T. W. Page, R. G. Guy, G. J. Popek, J.-F. Paris, and H. Garcia-Molina. Primarily disconnected operation: experience with Ficus. In *Proceedings of the Second Workshop on the Management of Replicated Data*, pages 2–5, November 1992.

[6] R. Housley, W. Ford, W. Polk, and D. Solo. Internet X.509 public key infrastructure certificate and CRL profile. Internet RFC 2459, 1999 January.

[7] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[8] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–65, July 1978.

[10] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 124–39, Kiawah Island, SC, USA, December 1999.

[11] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Fifteenth ACM Symposium on Operating Systems Principles*, pages 143–55, Copper Mountain Resort, CO, USA, December 1995.

[12] G. C. Necula and P. Lee. Safe kernel extensions without runtime checking. In *Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–43, Seattle, WA, USA, October 1996.

[13] B. Noble, B. Fleis, and M. Kim. A case for fluid replication. In *Network Storage Symposium*, Seattle, WA, USA, October 1999.

[14] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 288–301, Saint Malo, France, October 1997.

[15] J. I. Schiller and D. Atkins. Scaling the web of trust: combining Kerberos and PGP to provide large scale authentication. In *Proceedings USENIX Winter 1995 Technical Conference*, pages 93–94, New Orleans, LA, USA, January 1995.

[16] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–9, Austin, TX, USA, September 1994.