FuDiCo II: S.O.S. (Survivability: Obstacles and Solutions), 2nd Bertinoro Workshop on Future Directions in Distributed Computing, 23-25 June 2004, University of Bologna Residential Center, Bertinoro (Forlì), Italy.

A protocol family approach to survivable storage infrastructures

Jay J. Wylie, Garth R. Goodson, Gregory R. Ganger, Michael K. Reiter
Carnegie Mellon University

Abstract

A *protocol family* supports a variety of fault models with a single client-server protocol and a single server implementation. Protocol families shift the decision of which types of faults to tolerate from system design time to data creation time. With a protocol family based on a common survivable storage infrastructure, each data-item can be protected from different types and numbers of faults. Thus, a single implementation can be deployed in different environments. Moreover, a single deployment can satisfy the specific survivability requirements of different data for costs commensurate with its requirements.

1 Motivation

Survivable, or fault-tolerant, storage systems protect data by spreading it redundantly across a set of storage-nodes. In the design of such systems, determining which kinds of faults to tolerate and which timing model to assume are important and difficult decisions. Fault models range from crash faults to Byzantine faults and timing models range from synchronous to asynchronous. These decisions affect the access protocol employed, which can have a major impact on performance. For example, a system's access protocol can be designed to provide consistency under the weakest assumptions (i.e., Byzantine failures in an asynchronous system), but this induces potentially-unnecessary performance costs. Alternatively, designers can "assume away" certain faults to gain performance.

Traditionally, the fault model decision is hard-coded during the design of an access protocol. This traditional approach has two significant shortcomings. First, it limits the utility of the resulting system—in some environments, the system incurs unnecessary costs, and, in others, it cannot be deployed. The natural consequence is distinct system implementations for each distinct fault model. Second, all users of any given system implementation must use the same fault model, either paying unnecessary costs or accepting more risk than desired. For example, temporary and easily-recreated data incur the same overheads as the most critical data.

We advocate an alternative approach, in which the decision of which faults to tolerate is shifted from design time to data-item creation time. This shift is achieved through the use of a *family* of access protocols that share a common server implementation and client-server interface (i.e., storage infrastructure). A *protocol family* supports different fault models in the same way that most access protocols support varied numbers of failures: by simply changing the number of storage-nodes utilized, and some read and write thresholds. A protocol family enables a given storage infrastructure to be used for a mix of fault models and allows the number of faults tolerated to be chosen independently for each data-item.

We have developed a protocol family for survivable storage infrastructures [6, 8]. Each member of this protocol family exports a block-based interface. The survivable storage infrastructure consists of versioning storage-nodes that offer a single interface for all data, regardless of the fault model. Clients of the infrastructure realize a particular model for a data-item by interacting with the right number of storage-nodes.

Such a protocol family is particularly appropriate for increasingly-popular "storage brick"-based systems, such as Self-* Storage [5], Federated Array of Bricks [3], and Collective Intelligent Bricks [12]. Such systems seek to compose high-quality storage out of collections of cost-efficient, commodity servers. These storage servers are meant to be generic, and customers are meant to compose systems to fit their needs by simply buying a sufficient number of them. Using a protocol family assists this vision in two ways. First, one server software implementation can be reused for different environments: for example, a system on a secure machine room network might safely assume a synchronous model of communication, whereas a system on an open network susceptible to flooding attacks cannot safely make such an assumption. This capability allows for a single storage brick product to apply to many environments. Second, a single deployment can store data with differing survivability requirements without all data incurring the costs that must be paid for the most sensitive data.

Wide-area storage infrastructures, such as those proposed to run on network overlays, are another suitable environment for a storage protocol family. A wide-area infrastructure provides greater returns on investment if it can be reused for many purposes. A protocol family would allow different applications to specialize their use of the common infrastructure, rather than having to deploy their own infrastructure or pay unnecessary costs.

2 A survivable storage protocol family

To illustrate the concept, this section briefly overviews our protocol family, which has been designed for consistent access to redundant storage. Each member of the protocol family works roughly as follows. To perform a write, a client sends time-stamped fragments to the set of storage-nodes involved in storing the data being updated. Storage-nodes keep all versions of fragments they are sent until garbage collection frees them. To perform a read, a client fetches the latest fragment versions from the storage-nodes and determines whether they comprise a completed write; usually, they do. If they do not, additional fragments or historical fragments are fetched, until a completed write is observed (or, some family members may abort). Only in certain cases of failures or concurrency are there additional overheads incurred to maintain consistency.

Our protocol family is particularly interesting because it can be space-efficient, is optimized for the common case, and is scalable. Protocol family members can be space-efficient because they support *m*-of-*n* erasure codes (i.e., any *m* of a set of *n* erasure-coded fragments can be used to reconstruct the data), which can tolerate multiple failures with less network bandwidth (and storage space) than replication. Members are optimized for the common case: most read operations complete in a single round trip. Only read operations that observe write concurrency or failures (of storage-nodes or a client write) may incur additional work—failures ought to be rare, and concurrency is rare in file system workloads. Members are scalable since most protocol processing is performed by clients rather than servers. As well, read and write operations do not require server-to-server communication (a potential scalability bottleneck with regard to the number of faults tolerated).

2.1 Protocol family membership

There are four main parameters that differentiate members of the protocol family: the timing model, the storage-node failure model, the client failure model, and whether clients are allowed to perform repair.

Timing model. Protocol family members are either asynchronous or synchronous. Asynchronous members rely on no timeliness assumptions (i.e., no assumptions about message transmission delays or execution rates). In contrast, synchronous members assume known bounds on message transmission delays between correct clients/storage-nodes and their execution rates. In a synchronous system, storage-nodes that crash are detectable via timeouts, which provide useful information to the client. But, systems with assumed timeouts may produce incorrect results if unexpected delays occur.

Storage-node failure model. Our protocol family's members tolerate a hybrid failure model of storage-nodes. Storage-node crash failures, omission failures [13], or crash-recovery [1] failures can be mixed with Byzantine [10] storage-node failures. The concept of hybrid failure models was introduced in [14] with t crash failures mixed with $b \leq t$ Byzantine failures. For crash–Byzantine and omission–Byzantine members, t is the upper bound on storage-nodes that can crash and experience omissions, respectively. For crash-recovery–Byzantine members, t is the upper bound on the number of "bad" [1] storage-nodes (Byzantine storage-nodes are included in the set of "bad" storage-nodes). By setting $b = 0$ in each of the hybrid models, one gets a wholly crash, omission, and crash-recovery model.

Client failure model. Each member of the protocol family tolerates crash client failures and may additionally tolerate Byzantine client failures. Crash failures during write operations can result in subsequent read operations observing partially complete write operations. Readers cannot distinguish read-write concurrency from a crash failure during a write operation.

As in any general storage system, an authorized Byzantine client can write arbitrary values to storage, which affects the value of the data but not its consistency. Mechanisms are employed to ensure that writes by Byzantine clients have integrity (i.e., that every possible read operation will observe the same value) [7]. These mechanisms successfully reduce Byzantine actions to either being detectable or crash-like.

Repair by clients. Each member of the protocol family either allows, or does not allow, clients to perform repair. Repair involves having a client "write back" a value during a read operation to ensure that the value being returned is completely written. In repairable protocol members, all read operations can complete.

In systems that differentiate write privileges from read privileges, client repair cannot be allowed. Non-repair protocol members allow read operations to abort. Reads can be retried at either the protocol or application level. At the protocol level, concurrency is visible in the timestamp histories—an aborted read could be retried until a stable set of timestamps is observed. Other possibilities include requiring action by some external agent or blocking until a new value is written to the data-item (as in the "Listeners" protocol of Martin et al. [11]).

2.2 Selection of protocol members

Many, possibly conflicting, factors influence which protocol family member best meets a system's survivability goals. There are costs associated with "safer" members that may not be worth paying for some data. For example, asynchronous, Byzantine-tolerant members require more storage-nodes and more client computation than synchronous crash-tolerant members. As well, where and

Protocol family parameter	Example reasons
Synchronous	LAN, physically isolated systems
Asynchronous	WAN, resilience to DOS attacks on network
Crash-only	Closely monitored systems
Crash–Byzantine storage-nodes	Greater survivability, untrusted environments, complex software
Crash-only	Self-validating data, well-managed client systems
Byzantine clients	Critical data, open systems with many distinct clients
Repairable	NFS model of trusted client OSes
Non-repair	Distinct data publishers and readers

Table 1: **Example reasons for selecting protocol family members**

how the system is expected to be deployed impacts which members ought to be selected. With a protocol family, a deployed system using the “wrong” member could even migrate to the “right” member without changing the storage infrastructure.

Table 1 summarizes the properties that distinguish protocol family members, and lists example reasons for selecting certain member parameters. Synchronous members may be appropriate in isolated LAN environments, whereas in open environments, or in a WAN, asynchronous members may be preferred. There are many reasons to want to tolerate Byzantine storage-nodes; of equal interest though, is how many Byzantine faults (b) to tolerate and how many additional crash-recovery failures ($t-b$) to tolerate. If the goal is solely fault tolerance, $b=1$ and $t>b$ may make sense, whereas if security is the over-riding goal, $b=t$ may be preferred. Deciding whether or not to tolerate Byzantine clients could depend on either the type of data being stored or the client environment. The application domain of a given dataset is most likely the major factor in determining whether or not to employ a repairable member.

3 Related work and concepts

We describe and evaluate our protocol family in [6] and [8]. The performance and scalability of the asynchronous, repairable, Byzantine protocol member is evaluated in [7]. In a wholly Byzantine failure model, servers in the “Listeners” protocol of Martin et al. [11] broadcast new versions of data to clients that are listening; in our protocol clients ask for past versions of data. Frølund et al. have also developed a decentralized storage consistency protocol that accommodates erasure-coded data [4]; it operates in the crash-recovery failure model and does not tolerate Byzantine failures of clients or storage-nodes.

Previous researchers have explored configurable protocols and logically related protocol families, but we are not aware of prior protocol families that use a common infrastructure. Cristian et al. [2] systematically derive a logical set of atomic broadcast protocols for a range of fault models (from crash faults to a subset of Byzantine faults) in a synchronous environment. Cristian et al. use the term “family” to refer to the logical construction of the protocols (by layering protocols on top of one another) rather than their implementation. In our storage access protocol family, different members are realized in a common implementation (i.e., they share the common storage infrastructure) as well as being logically related. Hiltunen et al. developed a framework for building distributed services that are easily configured to handle different failures [9]. Again, the focus is logical modularity.

4 Summary

Replacing “niche” protocols suitable for a single purpose with protocol families allows a single implementation to be used in many environments and a single deployment to serve distinct requirements efficiently. The implementation of our read/write access protocol family for survivable storage demonstrates the concept’s feasibility. We have also had early successes developing a read-modify-write protocol family. Our promising initial results indicate that the protocol family approach to constructing survivable infrastructures is worthy of further exploration.

References

[1] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, **13**(2):99–125. Springer-Verlag, 2000.

[2] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: from simple message diffusion to Byzantine agreement. *Information and Computation*, **118**(1):158–179, April 1995.

[3] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. FAB: enterprise storage systems on a shoestring. Hot Topics in Operating Systems, pages 133–138. USENIX Association, 2003.

[4] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. A Decentralized Algorithm for Erasure-Coded Virtual Disks. Dependable Systems and Networks, June 2004.

[5] G. R. Ganger, J. D. Strunk, and A. J. Klosterman. *Self-* Storage: brick-based storage with automated administration*. Technical Report CMU-CS-03-178. Carnegie Mellon University, August 2003.

[6] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. *A protocol family for versatile survivable storage infrastructures*. Technical report CMU-PDL-03-103. CMU, December 2003.

[7] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. Dependable Systems and Networks, June 2004.

[8] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. *The safety and liveness properties of a protocol family for versatile survivable storage infrastructures*. CMU–PDL–03–105. Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, March 2004.

[9] M. A. Hiltunen, V. Immanuel, and R. D. Schlichting. Supporting customized failure models for distributed software. *Distributed Systems Engineering*, **6**(3):103–111, September 1999.

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

[11] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. International Symposium on Distributed Computing, 2002.

[12] R. Morris. Storage: from atoms to people. Keynote address at Conference on File and Storage Technologies, January 2002.

[13] K. J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, **SE–12**(3):477–482, March 1986.

[14] P. Thambidurai and Y.-K. Park. Interactive consistency with multiple failure modes. Symposium on Reliable Distributed Systems, pages 93–100. IEEE, 1988.