# An efficient protocol family for survivable storage infrastuctures

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

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 the access protocol.

This traditional approach has two significant shortcomings. First, it limits the utility of the resulting system—either the system incurs unnecessary costs in some environments or it cannot be deployed in harsher environments. The natural consequence is distinct system implementations for each distinct fault model. Second, all data stored in any given system implementation must use the same fault model, either paying unnecessary costs for less critical data or under-protecting more critical data. For example, temporary and easily-recreated data incur the same overheads as the most critical data.

We promote 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. 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 infrastructure of storage-nodes to be used for a mix of fault models and number of faults tolerated, chosen independently for each data-item.

We have developed a family of access protocols that exploit data versioning within storage-nodes to efficiently provide consistency for erasure-coded data. The protocol family covers a broad range of fault model assumptions (crash vs. Byzantine servers, crash vs. Byzantine clients, synchronous vs. asynchronous communication, client repairs of writes vs. not, total number of failures) with no changes to the client-server interface or server implementation. Protocol family members are distinguished by choices enacted in client-side software: the number of storage-nodes that are written and the logic employed during a read operation. Weaker assumptions require the use of more storage-nodes and additional computation for the client. Regardless of the assumptions made, the server implementation does not change.

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. 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 is efficient in three ways. First, all members support m-of-n erasure codes (i.e., $m$ of $n$ fragments are needed to reconstruct the data), which can tolerate multiple failures with less network bandwidth (and storage space) than replication. Second, most read operations complete in a single round trip: only reads that observe write concurrency or failures (of storage-nodes or a client write) may incur additional work. Third, most protocol processing is performed by clients, increasing scalability via the well-known principle of shifting work from servers to clients.

The poster includes an overview of the protocol family with an example of how it works. Graphs of response time and throughput measurements for up to nineteen storage-nodes are included. As well, there is a detailed breakdown of how client, server, and network resources are consumed by the protocol.