Group communication — still complex after all these years

Richard Golding
rgolding@us.ibm.com

Ohad Rodeh
orodeh@il.ibm.com

September 2, 2003

Abstract

Group communication is used in many file systems and storage controller systems. Ideally, there could be a single group communication system that serves the needs of all these various projects—software that was built once, tested once, and deployed everywhere. In reality, each project builds its own custom group services component. There are three reasons for this: most systems require only a few group communication features (but not the same ones); the scalability requirements differ by several orders of magnitude between systems; and most storage systems have unique programming environment requirements. These differences in requirements imply that group communication is a toolkit that needs to be built differently each time to fit the particular target system requirements.

To show to what extremes customization can go we take a look at two large-scale research storage systems: zFS and Palladio. These systems are interesting because they attempt to side-step the scalability boundaries of group communication by building mechanisms tailored to their systems, yet achieve strong semantics reminiscent of standard group communication systems.

1 Introduction

Many distributed and clustered storage systems are built using group communication systems (GCSes). These storage systems typically have high performance and reliability requirements. A GCS helps with implementing redundancy for failure tolerance, and with supporting many parallel components to get performance.

Some example storage systems that use group communication include:

- The IBM General Parallel File System (GPFS) [4], part of the SP^{TM} supercomputer, which uses the HACMP [2] programming suite.
- IBM TotalStorage SAN File System (aka StorageTank) [7], a Storage Area Network (SAN) file system, uses group services tailored for its metadata servers.
- IBM TotalStorage SAN Volume Controller [8], a SAN virtualization engine, uses group services custom made for its limited storage controller environment.

All these distributed storage systems use group communication, but using their own, custom-built implementations. This goes against one of the visions for GCSes: that they can provide a common communication infrastructure, implemented once and tested carefully. So why has it not worked out this way?

There are (at least) three reasons why people build custom group communication systems. First, these storage systems need specific semantics from their group mechanisms. Second, the intended scale of the systems varies widely: one system is focused on around ten group members, while another is focused on several thousands. Finally, each of these storage systems has its own programming environment.

It is our opinion that this trend will continue in the future. For storage systems, the cost of building a custom GCS is small compared to the cost of building the rest of the system, and the value of having a small, simple GCS that does not have extraneous

features is high due to the cost of testing and performance overhead.

This rest of this paper is structured as follows: Section 2 looks at some GCSes used in storage systems in more detail, showing how their requirements differ. Section 3 examines why GCSes are complex to build and require many design choices and customizations. Section 4 provides examples of two research projects using custom solutions.

2 Overview of GCS-based storage systems

HACMP [2] is targeted at highly available clustering applications. It provides the group services layer to the GPFS file system [4] at the core of the IBM SP^{TM} and ASCI White supercomputers, which scale from a few nodes up to several hundred. The group services allow cluster nodes to heartbeat each other, compute membership, perform multi-round voting schemes, and execute merge/failure protocols. When a node fails, a replacement node is chosen and responsibilities are passed to the new node. Lost tokens holding read/write locks and in-flight IO operations are recovered by the new node based on the new group view and on-disk state.

The TotalStorage SAN File System [7] contains a set of metadata servers responsible for all file system metadata — the directory structure and file attributes. The file system namespace is partitioned into a forest of trees, and the metadata for each tree is stored in a small database on shared disks. Each tree is managed by one metadata server, so the servers share nothing between them. This leads to rather light requirements for group services: they only need to quickly (100ms) detect when one server fails and tell another to take its place. Recovery consists of reading the on-disk database and its logs, and recovering client state such as file locks from the clients.

The TotalStorage SAN Volume Controller [8] is a storage controller and SAN virtualizer. It is architected as a small cluster of Linux-based servers situated between a set of hosts and a set of disks. The servers provide caching, virtualization, and copy services (e.g. snapshot) to hosts. The servers use non-volatile memory to cache data writes so that they can immediately reply to host write operations. Fault tolerance requirements imply that the data must be written consistently to at least two servers. This requirement has driven the programming environment in TotalStorage SAN Volume Controller: the cluster is implemented using a replicated state machine approach, with a Paxos-like [9] two-phase protocol for sending messages. However, the system runs inside a SAN, so all communication is through the FibreChannel, which does not support multicast, and messages must be embedded in the SCSI over FibreChannel (FCP) protocol. This is a major deviation from most GCSes, which are IP/multicast based.

3 Why group communication systems are complex

Many general-purpose GCSes have been built over the years. Some of them, such as Ensemble [6] and Spread [11], provide ways to customize the behavior of the communication to an application's needs. These systems are typically fairly complex pieces of software; why is that?

There are three things complex about the use of GCSes: semantics, scalability, and the target programming environment.

What kind of semantics should a GCS provide? Some storage systems only want limited features in their group services – such as StorageTank, which only uses membership and failure detection. Other systems want to communicate through the GCS, but they use the communication in specific ways – the SAN Volume Controller uses communication for a reliable replicated state machine. Some important semantics include:

- Ordering: FIFO-order, causal-order, total-order, safe-order. Can several kinds of ordering be used at the same time? If so, what are the ordering rules between, say, total-order messages and FIFO-order messages?
- Messaging: all systems support multicast, but what about virtually synchronous point-to-point?
- Lightweight groups: what are the guarantees between groups?

- Handling state transfer: what support is provided for handling failures, application state and state transfer?

- Channels outside the GCS: how is access to shared disk state coordinated with in-GCS communication?

Traditional GCSes have scalability limits. Roughly speaking, the basic virtual synchrony model cannot scale beyond several hundred nodes – but some products, such as GPFS on cluster supercomputers, push these limits. What pieces of a GCS are required for correct functioning and what can be dropped? Should we keep failure detection but leave virtual synchrony out? What about state transfer?

The product's programming and execution environment also bears on the ability to adapt an existing GCS to it. Many storage systems are implemented using special purpose operating systems or programming environments with limited or eccentric functionality. This is done in order to make products more reliable, more efficient, and less vulnerable to attacks. The TotalStorage SAN Volume Controller, for example, is implemented in a special-purpose development environment designed around its replicated state machine, and intended to reduce errors in implementation through rigorous design. Some important aspects of the environment are:

- Programming Language: C, C++, C#, Java, ML (Caml, SML)?

- Threading: What is the threading model used? Some possibilities are: co-routines, non-preemptive (where a thread needs to yield control of its own volition), in-kernel threads, user-space threads.

- Kernel: is the product in-kernel, user-space, both?

- Resource allocation: how are memory, message buffer space, thread context, and so on allocated and managed?

- Build and execution environment: What kind of environment is used by the customer?

- API: What does the GCS API look like: callbacks? socket-like?

This wide variability in requirements and environment makes it impossible to build one all encompassing solution.

4 Two specific examples

Below two examples show custom solutions to specific distributed problems in the storage domain. The solutions attempt to overcome the scalability limits of group communication by using other techniques. Group communication is still required to replicate and make highly available basic building blocks like security services, management consoles etc.

4.1 zFS

zFS [10] is a research project aimed at building a decentralized file system that distributes all aspects of file and storage management over a set of cooperating machines interconnected by a high-speed network. It is designed to be extremely scalable. It can be compared to clustered/SAN file-systems that use group-services to maintain lock and meta-data consistency. The challenge is to maintain the same levels of consistency, availability, and performance without a GCS.

zFS uses *object stores* (ObSes) [1] as storage media, leases for locking, and distributed transactions to guarantee consistency. It provides strong cache consistency and journaling of metadata operations to clients.

An object store is a storage controller that provides an object-based protocol for data access. Roughly speaking, an object is a file, users can read/write/create/delete objects. An object store handles allocation internally and secures its network connections. This provides a strong building block from which to build a file system. In zFS we also assume an object store supports one *major lease*. Taking a major lease allows access to all objects on the object store. The ObS keeps track of the current owner of the major-lease; this replaces a name service for locating lease owners.

zFS is a serverless file system containing only hosts and disks. The hosts run the management components as well as the file system clients. A host includes an in-kernel module that implements the VFS interface this makes zFS appear like a standard file system to user-space applications.

A zFS configuration can encompass a large number of ObSes and hosts. While hosts can fail, we assumes ObSes do not. This builds on a large body of work invested in making storage-controllers highly reliable. In zFS, if an ObS fails then a client requiring a piece of data on it will wait until that disk is restored. zFS assumes the timed-asynchronous message-passing model.

Files and directories in the file system map to objects in the object store. An ObS does not distinguish between files and directories both appear as regular objects in its view. A directory contains an array of records where a record contains a file name and its location in the system: a tuple of ObS-id and object-id. The root directory is at a fixed known location allowing each client to traverse the file system hierarchy autonomously.

zFS uses leases to manage access-control from clients to data. The description here is a simplification, the full details are given in [10].

A lease-manager (LMGR) takes the major-lease for an ObS. It gives hosts read/write leases per extent using a strict cache-consistency policy. When a host accesses part of a file F on ObS α it first requests the appropriate lease from LMGR$_\alpha$. A triangle is formed, the ObS gives a lease, manipulated by the lease-manager, and given to the client. With the lease, the client can then access the ObS directly. This allows clients to perform IO directly to the disk.

When a host wishes to locate an LMGR for ObS α it queries α. If no LMGR is currently assigned, then the host takes upon itself the role of LMGR$_\alpha$. No name service is needed for locating LMGRs.

LMGRs are located on hosts, and can therefore fail. When LMGR$_\alpha$ fails it's major-lease will still hold for a while. After the major-lease expires another host can take upon itself the role of LMGR$_\alpha$. All hosts that hold leases on α have sufficient time to flush their dirty data. This makes lease-managers state-less, they can fail at any time and the system will continue to function.

The most challenging part of this architecture is performing metadata operations. All metadata operation: create file, delete file, create directory etc., are distributed transactions spanning several object stores. For example, file creation should be an atomic operation. However, in reality it is implemented by creating a directory entry, then creating a new object with the requested attributes. A host can fail in between these two steps leaving the file system inconsistent. A delete operation involves first removing a directory entry then removing an object. A host performing the delete can fail midway through the operation leaving dangling objects that are never freed, which will cause loss of disk space. Rename is the most difficult operation.

4.2 Palladio

The Palladio system [3, 5] is a project to build a scalable, high-performance block storage system. It is designed to be built from many relatively small machines connected by high-speed IP networking. The nodes might be placed at multiple sites to support recovery from site failures. The system is intended to work well at ten nodes, and scale up to storing petabytes of data across thousands of nodes.

Data in Palladio is organized into *virtual stores*. Each virtual store presents an array of blocks, much like a disk or logical volume. Virtual stores are independent: IO operations done on one store have no effect on the data in other stores (other than the performance effects of contention for shared resources such as the network.)

Hosts using the Palladio system use a local storage driver to perform IO operations. The driver talks to two other services: *storage devices* and *managers*. A storage device is a container for several chunks, each of which holds some data. Chunks from several storage devices are aggregated together using a *layout policy* to hold all the data in the virtual store. Example layouts include striping, where the data from the virtual store is spread over multiple chunks non-redundantly, replication, where the data is replicated identically between two or more chunks, and RAID layouts, which combine striping with some form of computed redundancy code. The manager service for a virtual store acts as the contact point where

clients can get a copy of the layout, and coordinates changes to the layout amongst all the storage devices involved.

The system is designed as three tiers, each building on the ones below it: IO processing, layout management, and global system control. The IO processing tier has the highest performance requirement but also the most localized view for each operation, while the global system control has modest performance requirements and the most global view.

- IO processing. This tier handles each IO request that hosts make. Each IO operation must be performed quickly, and large numbers of them must be processed in parallel – even to data that are shared, such as occurs when reads and writes are being done to data that are being moved from one storage device to another. It is implemented as a lightweight transaction protocol that borrows heavily from group communication. A write that spans multiple storage devices, for example, is processed atomically and with total ordering. Most IO operations will span up to around ten storage devices, with two being most common. Rare IO operations can span hundreds of storage devices.

  This protocol differs from traditional group communication in a few ways. First, each IO operation can (and typically does) involve a different set of storage devices, computed from the virtual store, offset and length of the request, and the layout. Each of these operations is independent of other operations, but must be consistently ordered. Second, different data are usually delivered to each of the devices (consider striped layouts). Finally, the transaction protocol is optimized for non-failure cases. When a failure occurs, the protocol simply blocks until the problem is detected and resolved by the next tier.

- Layout management. This tier keeps the layout for each virtual store consistent and correct amongst the storage devices and manager holding parts of that store. Each virtual store is independent. This protocol borrows from group membership protocols: managers and storage devices heartbeat each other to detect probable failures, and when membership changes occur, an atomic consensus mechanism ensures that all storage devices and managers see a consistent view. Membership changes are, of course, ordered with respect to data operations. Typical operations involve tens of participants, with a maximum around a hundred. The layout management tier only implements the mechanism for changing the layout; it reports suspected failures to the global control tier, and makes changes based on its responses.

  This protocol differs from GCS membership protocols primarily in that it handles manager failure specially: when a manager for a virtual store fails, the protocol elects a replacement from a pool of nodes that can host manager services.

- Global system control. This tier reacts to node failures, creation and removal of virtual stores, and measurements of workload, allocating storage resources in order to balance load and maintain needed redundancy. Some of these events come from user actions, while others come from the failure detection mechanisms in the layout management tier. This requires a global view, but a global view over several thousands of nodes at multiple sites is not feasible. Instead, hierarchical decision mechanisms are used to divide the system into regions of feasible size.

The IO processing and layout management tiers are thus similar in purpose to a general-purpose GCS, but are specialized to the problems at hand. This specialization results in a good fit with the semantic needs of the storage system: the lightweight transaction mechanism, for example, ensures that feasible recovery is simple after most kinds of failure.

4.3 Comparison

It is difficult to compare zFS and Palladio as this isn't an apples-to-apples comparison. zFS is a file-system, Palladio is a block-storage system. zFS uses object-stores where Palladio uses "almost" regular disks. It is interesting to compare the systems nonetheless since they are both highly-scalable and attempt to side-step the boundaries of group-communication.

Below, we simplify and and assume that both systems use "disks".

Palladio:

- Uses group-services outside the IO path.
- Performs IO to a small set of disks per IO
- Uses light-weight transactions
- Does not use reliable disks
- Uses a name-server to locate internal services

zFS:

- Does not use group services.
- Performs IO to a small set of disks per transaction
- Uses full, heavy-weight, transactions
- Uses reliable disks
- Does not uses an internal name-server

It remains to be seen if zFS will manage to do without group-services altogether.

5 Summary

It is the authors' opinion that group communication is complex and is going to remain complex, without a one-size-fits-all solution. There are three major reasons for this: semantics, scalability, and operating environment. Different customers (1) want different semantics from a GCS; (2) have different scalability and performance requirements; and (3) have a wide range of programming environments including varying operating systems, threading models, memory footprint requirements and more. We believe that in the future users will continue to tailor their proprietary GCSes to their particular environment. The good news is that the field is going to continue to be challenging, the bad news is that we are going to be solving variants of the same problem over and over again.

References

[1] `www.snia.org/tech_activities/workgroups/osd`

[2] *Group Services Programming Guide and Reference, RS/6000 Cluster Technology*. IBM, International Technical Support Organization, 2000.

[3] K. Amiri, G. Gibson, and R. Golding. Highly concurrent shared storage. In *International Conference on Distributed Computing Systems*, April 2000.

[4] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *First Conference on File and Storage Technologies (FAST)*, January 2002.

[5] R. Golding and E. Borowsky. Fault-tolerant replication management in large-scale distributed storage systems. In *Symposium on Reliable Distributed Systems*, April 1999.

[6] Hayden, M. The Ensemble system. Phd Thesis TR98-1662, Cornell University, Computer Science, 1998.

[7] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. Storage Tank, a heterogeneous scalable SAN file system. *IBM Systems Journal*, 2(42), 2003.

[8] J. S. Glider, C. F. Fuente, and W. J. Scales. Software architecture of a SAN storage control system. *IBM Systems Journal*, 2(42), 2003.

[9] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[10] O. Rodeh and A. Teperman. zFS – a scalable distributed file-system using object-disks. In *Goddard Conference on Mass Storage Systems and Technologies*, April 2003.

[11] Stanton, J. and Amir, Y. The Spread wide area group communication system. TR CNDS-98-4, Center for Networking and Distributed Systems, Computer Science Department, Johns Hopkins University, 1998.