DataMesh, house-building, and distributed systems technology

Position paper for 5th ACM SIGOPS European Workshop

John Wilkes

wilkes@hpl.hp.com

Concurrent Systems Project
Hewlett-Packard Laboratories

15 April 1992

1 System design reconsidered as architecture

The past several years have shown a steady increase in knowledge about building (distributed) computer systems. Techniques like (L)RPC, scheduler activations, (relatively) machine-independent memory management, ordering and causality, and distributed caching and consistency protocols have developed to the point where we have a useful array of tools at our disposal. There are doubtless many improvements still to be made in such techniques: more tools will be developed, existing ones will be sharpened. But the biggest opportunity available to us is in learning how to apply the tools we have.

Consider the analogy of a distributed computer system as an ensemble of buildings. Rooms might represent address spaces; their occupants, threads; buildings the nodes of the distributed system.

RPC systems are like sending paper messages from one building to another by postman (on roller skates, for the faster ones). Networking technology has developed smooth, straight paths between buildings, plus a structure of intersections that mean that the entire complex doesn't have to be paved over, but the handoff between the postman and the building inhabitants still leaves much to be desired: we still struggle through multi-ported airlocks, and an army of intermediaries.

Causality and ordering mechanisms use audit trails on the pieces of paper sent between rooms to determine when to deliver a new one to the occupants of a room. Just like in the real world there is a team of people busy installing new sewer systems, power lines, and roadways between the buildings: with commendable enthusiasm they sever communication paths, and disable entire buildings by putting pickaxes through a power line, and generally cause mayhem for the inhabitants—all in the name of progress, of course!

So far we (the distributed OS community) have been concentrating almost exclusively on the construction of ever-better building components, without attending with due diligence to the architecture of the whole—or even the makeup of some of the individual buildings.

We have assembled an impressive set of components, but as yet relatively little guidance on how to apply them: there are precious few cohesive schools of architecture, or even plans for buildings that perform specific tasks well. Rather, we have concentrated on giving our doorways Teflon linings; on embellishing the decoration around the window panes; and insisting that all the rooms are fitted with precisely the same set of amenities. And recently, we've been trying to propose that every building should be constructed as a single floor of identical rooms without doors, situated on top of a tiny cramped basement (a microkernel).

The time has come to develop—and publicize—more endeavors that stretch the architects' art, rather than the plumbers'.

We all do architecture to some degree: the thrust of this proposal is that we should work harder as a community to discuss and describe the good and the bad, so that we can all improve. We've got a very good set of tools and components for constructing distributed systems, but not enough guidance on applying them.

2 Dedicated buildings and systems

One architectural technique that has received insufficient support in the systems arena is the art of building for a purpose: developing a design (based on standard components, but extending them where needed) that is much better at performing a particular function than a general-purpose design could be. A railway station isn't the same kind of building as a housing complex—and a single design that tried to accommodate both would be unlikely to be successful.

In the distributed systems arena, we are well placed to exploit this specialization: our buildings (computers) are connected only by relatively narrow pathways (networks), and the internal construction of any one building is not visible to the outside. This freedom is relatively new: combined with the increases in understanding about individual components of distributed systems technology, we have an unparalleled opportunity to develop—even exploit—this in ways that will provide better (faster, more scalable, more fault-tolerant) systems.

Fortunately, computer systems are not buildings: they don't have to be hand-crafted for each individual use and plot of land on which they must sit. Instead, a relatively few general designs can serve the needs of a range of needs.

The rest of this position paper is devoted to an overview of one such specialized design—itself a distributed system. Although there are interesting advances in the technology of individual components, the emphasis here will be on the manner in which the existing knowledge in distributed systems is being applied to the overall approach.

3 The DataMesh storage system

In many computer systems, a significant amount of system resources are taken up with storing and retrieving data from persistent memories—disks, tertiary storage, and so on. Recent improvements in networking (such as gigabit-speed links) make it possible to offload storage management to a specialized server, and we believe that by doing so much better performance can be achieved than would otherwise be the case.

Our target environment is a shared storage server for a collection of clients—perhaps workstations, perhaps nodes in a parallel database or computation engine. Our goal is to develop a system storage architecture to provide high performance, high availability, scalability (in both size and component type), and standards-based connections to the open systems environment. We believe that these requirements in turn suggest particular architectural solutions:

- *high performance*: the use of parallelism and the close coupling of processor power with storage elements;
- *high availability*: no single points of failure (i.e., there must be built-in redundancy), coupled with fault tolerant software;
- *scalability and long life*: a modular architecture, to allow a DataMesh server to expand and adapt to changing requirements over time, with smooth incremental growth;
- *different kinds of storage service*: internally specialized components that allow mixing and matching of parts to achieve the right balance for a particular application configuration;
- *open systems interconnection*: the ability for a DataMesh server to be attached to the outside world through several hardware and software interconnect standards.

The themes explored in the discussion that follows are three-fold: applying the theme of internal diversity for flexibility at many different levels in the DataMesh architecture; application of a set of existing distributed systems techniques; and extending some of these as a result of architectural considerations.

3.1 Internal diversity

In contrast to those who believe in general-purpose solutions for all eventualities, we believe that "divide and conquer" is an important technique—indeed, the very notion of specialized systems being espoused here is one instance of this approach. We carry this principle into the design of the DataMesh itself.

For example, we don't believe that a file system designed for small objects with clustered accesses to them (such as a 4.2BSD file system) is appropriate for storing multimedia objects with strict performance constraints on bandwidth and timeliness. Nor do we believe that everybody needs exactly the same degree of reliability or availability for their data—and cost and performance tradeoffs can usefully be made to exploit these differences.

We reflect this diversity in many ways: in the specialized hardware nodes in a DataMesh; in the overall system architecture (*Jungle*), which is a framework for providing a multitude of different design alternatives; in the differentiation of functions within even the lowest logical layer of the Jungle framework (virtual devices); in a variety of different *policies* that implement those design alternatives.

Our chosen hardware solution is an array of nodes of various types (Figure 1):

- *port nodes* provide connectivity to the outside world through I/O interfaces like SCSI, or LANs like FDDI;

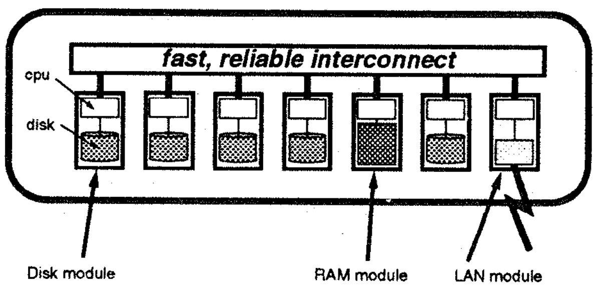

Figure 1. DataMesh system hardware model.

- *disk nodes* provide secondary storage;
- *RAM nodes* (volatile or non-volatile) are used for data and metadata caching, read-ahead, and write-behind;
- and *tertiary storage nodes* allow expansion to high total capacity at low cost (e.g., an optical jukebox, R-DAT tape, or robot tape library).

All the nodes are linked by a fast, reliable, small-area network that is internal to the DataMesh—and can thus be specialized for low latency and high bandwidth without the compromises necessitated for local area networks. The whole ensemble is programmed so that it appears as a single storage server to its external clients.

An orthogonal specialization occurs in the overall software architecture for DataMesh—which is called Jungle. The lowest Jungle level is a smart *chunk store*, or set of *virtual devices*. These hold raw bags of bytes, on top of which there is a layer of *chunk vector managers* that provide an abstraction composed of sequences of chunks. Finally, there is a layer of *Jungle-thing access managers* (JTAMs) that provide application- and system-specific interfaces to the stored data.

These layers are separate to allow for different implementations and policies to be applied at each layer. For example, multiple JTAMs may make use of a single kind of chunk vector; and different chunk vector implementations will exist for different performance tradeoffs (short random I/O versus highly predictable sequential transfers, for example).

Jungle software will run on both DataMesh server and client nodes; distributed cache management will concern itself with system-wide memory management, balancing alternate needs. We are thinking of experimenting with a variety of competitive memory economies [Waldspurger92] to achieve this.

Our programming model follows that of the proxies of [Shapiro86]: any access to an object is through a local copy of the manager code. Although this may sound a little extreme at first sight, we also allow the local manager code to be but a shadow of the real thing—for example, a stub (with or without some local caching), that just forwards all requests to a remote copy of the full manager code.

3.2 Scalability and performance

The Jungle framework is a very high-level structure. A more detailed and concrete example of our use of internal diversity is provided by the first phase of the DataMesh work, which is concentrating on storage servers that support operations on fixed-size blocks of data. This interface corresponds almost exactly to the Jungle virtual device layer; for prototyping purposes, we have chosen to test our work by making the interface accessible through SCSI connections that emulate regular disk drives (plus a few extensions).

There are two conflicting needs here: scalability to large numbers of nodes, and the need to maintain moderate amounts of fast-changing state to extract the maximum performance from the disk nodes. The former is a result of our desires to design a system capable of scaling up to a couple of hundred disk drives. The latter is a consequence of our indirect-disk technology, which provides very fast writes—at the cost of an indirection table held in RAM that is updated on every write [English92]. Unfortunately, the two needs are at odds with one another: no single general-purpose technique can do both efficiently.

The architectural solution we have adopted is (once again) to specialize: we layer the problem into distinct components (see Figure 2), and apply well-known techniques at each to achieve both our objectives.

Multiple hosts can be connected to a single DataMesh through one or more *ports* (e.g., SCSI channels, gigabit links). Port-level communications are managed by *spigot* software—channel striping and routing is done here. An incoming request is passed on to a *dealer*, which uses static partitioning policies (so it can operate at high speed) to disperse it across one or more *decks*. The decks do dynamic load balancing, and hold Loge indirection tables. Finally, data is stored on *cards*, which contain embedded disks and perform rotation-based placement optimizations.

By making dealers responsible for static load balancing (we use hashing, loosely based on the style of the VAXcluster lock manager) [Kronenberg87], we are able to make a system scale to large numbers of nodes without requiring dealer-to-dealer communication. On the other hand, the state information in the decks allows

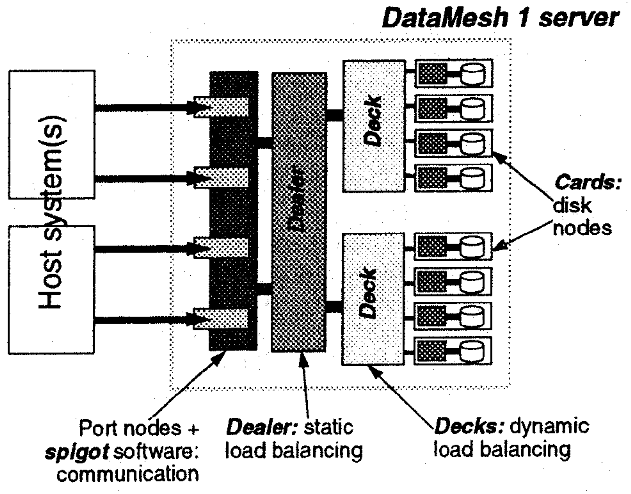
DataMesh 1 server
Host system(s)
Dealer
Deck
Deck
Cards: disk nodes
Port nodes + spigot software: communication
Dealer: static load balancing
Decks: dynamic load balancing

Figure 2. DataMesh phase 1 logical architecture.

to make dynamic performance optimizations across a much smaller number of disks—and our simulations suggest that most of this type of benefit is to be had from a half dozen or so devices.

3.3 Causality extended to disk data

The state information in the decks gives us considerable performance advantages; however, it is also a potential point of failure so it is important that mechanisms exist to rebuild it if something goes wrong. Driven by this architectural need, we extended the Loge disk technology to greatly improve the recovery and fault-tolerance properties provided. We did this by exploiting the shadow-writing nature of Loge disks to extend causality and ordering right down onto the disk. By contrast with most existing work has concentrated on applying these techniques to main-memory data structures.

We are able, as a result, to offer a great many benefits that fall out of our ability to reconstruct the ordering of events. Two powerful examples are the ability to recovery automatically to consistent on-disk states after a power failure or crash, and the removal of almost all synchronous I/Os.

The details of the mechanisms are presented in a technical report [Chao92]; for now, the important point is that we have found novel results from applying a well-understood technology (causality) in a new way as a result of a higher-level need. This is a direct (and satisfying!) example of the architectural approach espoused in this paper.

3.4 Interconnect

A subject dear to the heart of almost any distributed systems designer is the performance and structure of the communication layer. We found that our architectural needs led us to a three-pronged approach: bulk data traffic for moving data around the system without looking at it more than absolutely needed; normal rpc for control interactions; and—and this seems to be relatively new—a low-cost mechanism for performing performance optimizations.

When a request arrives at a DataMesh, the amount of time available for deciding what to do is relatively small: we budgeted ourselves about 1ms total overhead for deciding how to handle the query. We expect that one or more nodes may have copies of (some of) the data, so we'd like to find these as quickly as possible. If no node has the data cached in RAM, we'll have to go to disk—but again, we'd like to do an optimization decision based on which copy is going to be the fastest to access.

Suppose that we need to probe 10 nodes to find the best copy of the data, followed by a normal rpc to initiate the transfer. If we budget 0.3ms for the rpc, that leaves us 0.7ms for the remote probes and other optimization decisions—or about 50μs total for each probe, once we allow for a few cycles for the decision-making and bookkeeping. To achieve this, we had to invent a new kind of ipc operation: one that gave us a very quick indication of the cost of accessing a chunk of data. The price we chose to pay was to allow it to be wrong: i.e., we exploited the nature of hints that are so useful in distributed systems. The point here is that such a trade-off is unlikely to occur spontaneously from a prototol design effort: it occurred to us because we had a design in our minds for the larger system.

4 Conclusion

A primary function of a building architect is to determine the best design to serve the needs of the client. Such designs use many standard components; engineering and physics determine what can and cannot be done. But the skill of composition, and the molding of existing designs to suit the purposes of the client, are what determines whether or not the architect is successful, rather than the choice of the very latest kind of air conditioning system or wall cladding. (Of course, advances in air conditioning systems may make some new design easier to build, or reduce a building's cost, so such technology changes have to be constantly tracked and exploited—as tools, not as driving forces for designs.)

This position paper has argued that the distributed systems field has reached the point where we can (and should) be increasing the emphasis on the architectural side of our work. It has further advocated that a suitable vehicle for doing this is the field of specialized systems, which are targeted towards performing a few tasks extremely well. These will prove a much more exciting arena than the continuing commonality of "general purpose systems", because the metrics for success are clearer, the environment more narrowly focussed, and new technologies easier to test and improve in isolation. An example of this approach is the DataMesh project, which is applying much of the known distributed systems technology to developing a high-performance, highly functional storage server, with encouraging results.

References

[Chao92] Chia Chao, Robert English, David Jacobson, Alex Stepanov and John Wilkes. *Mime: a high performance storage device with strong recovery guarantees.* CSP technical report HPL–CSP–92–9, Hewlett-Packard Laboratories, March 1992.

[English92] Bob English and Alex Stepanov. Loge—a self-organizing disk controller. *Proceedings of Winter USENIX'92* (San Francisco, CA) Jan. 1992.

[Kronenberg87] Nancy P. Kronenberg, Henry M. Levy, William D. Strecker, and Richard J. Merewood. The VAXcluster concept: an overview of a distributed system. *Digital Technical Journal* 5:7–21, September 1987.

[Seltzer90] Margo Seltzer, Peter Chen and John Ousterhout. Disk scheduling revisited. In *Proceedings of the Winter 1990 USENIX* (Washington, DC), pp. 22–26 January 1990.

[Shapiro86] Marc Shapiro. Structure and encapsulation in distributed systems: the proxy principle. In *Proceedings of 6th International Conference on Distributed Computing Systems* (Cambridge, Mass), pp. 198–204. IEEE Computer Society Press, Catalog number 86CH22293-9, May 1986.

[Waldspurger92] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and W. Scott Stornetta. Spawn: a distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–17, February 1992.

[Wilkes89] John Wilkes. DataMesh — scope and objectives: a commentary. DSD technical report HPL–DSD–89–44, Hewlett-Packard Laboratories, July 1989.

[Wilkes91] John Wilkes. DataMesh — parallel storage systems for the 1990s. *Proceedings of the 11th IEEE Mass Storage Symposium* (Monterey, CA), October 1991.

[Wilkes91a] John Wilkes and Raymie Stata. Specifying data availability in multi-device file systems. *Position paper for 4th ACM-SIGOPS European Workshop* (Bologna, 3–5 September 1990). Published as *Operating Systems Review* 25(1):56–9, January 1991.