WAFT: Support for Fault-Tolerance in Wide-Area Object Oriented Systems

Lorenzo Alvisi, University of Texas at Austin
Keith Marzullo, University of California at San Diego

July 22, 1998

The difficulties that wide-area networks present to application designers are severe. For example, wide-area networks suffer from very unpredictable communication properties, which make it hard to distribute a computation over multiple sites. This is made even harder because what constitutes poor communication depends on the application's quality of service requirements. Yet, two major attractions of wide-area systems are the ability to support geographically dispersed cooperative work (for example, [15]) and the possibility of harnessing many processors for massively parallel computation (for example, [18, 14]). Both of these attractions push one to design geographically-dispersed applications, and therefore one must address the problem of unpredictable communication properties.

In addition, wide-area networks are not secure environments, and so the applications that are designed to run in them must be able to withstand security attacks. Replication, which is needed for fault tolerance, is an obvious point of attack. Rollback-recovery protocols, for instance, use replication in time [10] by restoring crashed agents to a previous state and repeating lost execution. A malicious user who alters the information used during recovery can affect the state to which a failed object is restored, thereby introducing a Trojan horse. The problem of keeping recovery information secure is especially acute for techniques such as causal logging [3, 4, 5] because causal logging replicates this information in the volatile memory of multiple processes. Nevertheless, the tampering of recovery information used by rollback-recovery protocols is not a well-studied topic in the fault-tolerance community.

The tools that are being used to build wide-area distributed systems are inadequate. Some tools (for example Legion [12]) attempt to hide the complexity introduced by wide-area communications. They attempt to present the illusion of a single computational environment, which deprives the designer of the ability to structure an application with the wide-area network in mind. Other tools (for example, Electra [13]) are very complex and are based on techniques, such as active replication, that are ill-suited for wide-area execution. [19]

There have been a few examples of tools that have been built specifically to support the construction of applications for wide-area networks. For example, Ahamad, Raynal and Thia-Kime are developing protocols [2] for implementing replication over a wide-area network based on causal consistency [1], Babaoglu and Schiper are developing support for active replication over wide-area networks [7], and in the Eternal project Narasimhan, Moser and Melliar-Smith are developing replication strategies for wide-area networks encapsulated in a CORBA framework [17]. These tools only address a part of the problem of building applications in wide-area networks. The tools of [2] work only for applications for which causal consistency is sufficient, and their notions of quality of service are only partially developed. The tools of [7] are based on active replication. Active replication requires tighter synchronization among the replicas than do passive replication techniques, and so they are ill-suited for an environment in which communications latency can be high. None of these three projects address the protection of the fault-tolerance mechanisms against security attacks.

In the DARPA-funded WAFT project, we are conducting research into the design and implementation of tools for building applications in wide-area networks. Specifically, we intend to:

- Develop service replication strategies appropriate for wide-area networks. Such strategies must be able to adapt to changing communication properties and able to withstand security attacks.

- Generalize the facility of *failure detectors* to agreement on the lack of required quality of service. Such a facility allows the responsibility of recovery to be shifted from servers to clients. This is necessary since, in general, only clients can determine how to recover from a reduction in quality in service.

- Embed the resulting services in an distributed object computation system such as Legion or CORBA. It is important that this embedding be done so that the complexity arising from building wide-area applications can be contained rather than spread throughout the application's code.

To motivate the set of functions that we believe are needed for building applications in wide-area networks, we first briefly describe one such application, the Nile system [18]. Nile provides a distributed environment for the execution of embarrassingly parallel programs written by high-energy physicists of the CLEO collaboration [9]. These programs can process gigabytes of data and can require days of execution time to complete. Based on the available computational cycles and the replication patterns of the data, the control system of Nile breaks apart these large jobs into subjobs that run in parallel. The Nile control system chooses the site on which each subjob will run based on the subjob's communication requirements and the locations of the data that it consumes. There is an independent copy of the Nile control system running at each site. When fully deployed in 2000, Nile will run in a network of approximately thirty sites, with each site having from tens to hundreds of machines. The sites will be spread across North America and connected using the Internet.

Nile is implemented using CORBA and both C++ and Java. The Nile control system consists of tens of relatively long-lived objects and up to approximately 100 relatively short-lived objects. Each subjob of a parallelized job is encapsulated in a CORBA object. There may be tens of thousands of these objects at any time. Most Nile control system objects communicate with several other Nile control system objects in a non-hierarchical manner, while subjob objects communicate with only a few objects in the Nile control and data delivery systems.

Low-Cost Object Resiliency It is difficult with current tools to achieve object availability at a reasonable cost. The time to develop the necessary software can be daunting and the performance of the resulting system can be poor.

For example, Nile objects are critical to the users who have jobs running. We began implementing Nile on top of the Electra+Isis orb, thereby using active replication for fault-tolerance. Each object in the Nile control system was replicated a few times within the same site. After a year of struggling to make this approach work, we have abandoned Electra+Isis.

One problem with Electra+Isis is that it is very hard to use. Most of the Nile control system objects are straightforward, and could be implemented quickly except for the function required to support active replication under Electra. In many cases, the time needed to develop and debug the fault-tolerance function was at least five times as much as was required to develop the core function. Some of the complexity arose because Electra, which was developed as a university research project, is not commercial-grade software. Much of the complexity arose, however, from Isis. In order to achieve reasonable performance from an Isis application, the more complex Isis primitives must be used in clever ways. This complexity is compounded by having the Isis primitives embedded into the CORBA model. [19].

Furthermore, the availability requirements of Nile are not as stringent as to require active replication. Crash failures are rare, and having a crashed object recover in tens of seconds is acceptable. In addition, the demand being placed on the Nile control system is not so high that multiple objects processing requests in parallel are required to keep up with this demand. Hence, Nile does not benefit from active replication.

Nile is not unusual in its availability requirements. Very high availability requires active replication, but active replication requires fairly tight synchronization. Application semantics must be exploited to loosen these synchronization demands, thereby adding complexity. We believe that an approach based on passive replication can provide sufficient availability for the vast majority of applications. In addition, the looser synchronization demands of passive replication make it less important to exploit application semantics to improve performance.

Partitions and Quality of Service In common terminology, one says that an object a is partitioned from an object b when the network between a and b fails to deliver data from b to a. We believe that one should instead define a partition as when a requires a better quality of communications from b than the network is currently providing. That is, the application has the ultimate say in when the system declares that a and b have partitioned.

We illustrate this difference by using a simple example from Nile. Recall that a job j is split into perhaps hundreds of subjobs. Each subjob is represented by a *Subjob* object, and all of j's Subjob objects are managed by a single *Job Manager* object for j. Furthermore, in one design of Nile, a *Collector* object waits for methods invocations from each of the Subjob objects. [14] These method invocations are used to send incremental results as they are computed by a subjob. The collector can then combine and display these partial results to the user who submitted the job. Subjobs execute at a reasonably constant rate, and so the user has an implicit soft real-time requirement on the accumulation on the partial results. The Collector object can use this soft real-time requirement to decide whether the communication between the subjob and the Collector object is so poor that a partition should be declared and the subjob should be restarted at a new site.

The Job Manager object must agree with the Collector object that there is a partition because the Job Manager is responsible for restarting the Subjob object. In this particular case, this poses no difficulties because the Subjob object does not invoke any methods on the Job Manager object. In general, however, an agreement protocol is required. Suppose that three objects a, b and x are communicating with each other. Let a decide that the quality of service it is receiving from x is insufficient, and therefore decide to stop communicating with x in favor of a fourth object y. If a does not synchronize its switch from x to y with b, then b may find itself simultaneously responding to invocation from x and from a that has decided that x is unavailable. This is exactly analogous to the *failover* problem in primary-backup protocols [8] and can be solved in asynchronous systems by using a kind of agreement protocol commonly used in group membership protocols [20].

Hence, there are situations in which a decision to adapt to poor quality of service requires an agreement protocol. In other cases, adaptation can occur with a simpler protocol. Deciding when running an agreement protocol is necessary before adaptation is an open problem that we intend to address.

Lazy Active Replication Replication can both improve and reduce availability. It can improve availability because it maintains multiple copies of objects to mask failures. It can reduce availability because these replicas may need to synchronize with each other, which may slow down the service.

For example, the Nile Scheduler object decides how to parallelize a job into subjobs and allocates subjobs to sites based on the location of the data that the job references and the availability of computational resources in the different sites. One way to implement this scheduler is to use active replication with a small number of replicas all located within the same site S. This solution is attractive for its simplicity, but it allows the service that the scheduler provides to become unavailable to objects in another site S' when S' partitions away from S. An alternative solution that addresses this limitation places a replica at each site. Assuming that partitioning can occur between any pair of sites, placing a replica at each site ensures that a global scheduler is available to any object no matter how the network partitions. If one wishes the service to behave identically to the single-site solution, then the replicas must use atomic broadcast to mutually update their states. Atomic broadcast, however, does not scale well with the number of recipients, and so any realistic solution that uses this degree of replication would instead need to be satisfied with less consistency than what atomic broadcast provides [14].

One might try to address this scaling problem by creating replicas only when they are needed. For example, one could create a global scheduler replica g' in site S' only when it partitions away from S. Ideally, one would create g' with the same state that the global scheduler g in S held immediately before the partition. Unfortunately, one cannot usually predict partitions and so this approach is unimplementable. We can, however, create g' with a state that is indistinguishable by the objects in S' from the state held by g when the partition occurred. Let g_x be the latest state of g that casually precedes the state of some object x. For all such x in S_1, we ensure that the state to which g' is recovered does not causally precede the state of g_x. We call this approach *lazy active replication*, since it creates a replica only when needed, yet the application is as consistent as if it had been created and updated using active replication before the partition occurred. In fact, the consistency provided by lazy active replication is very similar to the consistency provided by partitionable group membership protocols such as [11, 16, 6].

For many applications, this degree of consistency is sufficient. For example, in the case of the Nile scheduler, the state that g' will be given will reflect all scheduling decisions affecting clients and resources in S'. Furthermore, lazy active replication achieves this without incurring the synchronization costs of active replication. Lazy active replication can be thought of as a form of passive replication based on replication in time, and is therefore well-suited to use in a wide-area network.

References

[1] M. Ahamad, P. W. Hutto, G. Neiger, J. E. Burns, and P. Kohli. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[2] M. Ahamad, M. Raynal, and G. Thia-Kime. An adaptive protocol for implementing causally consistent distributed services. Available on http://www.cc.gatech.edu/fac/Mustaque.Ahamad/pubs.html.

[3] L. Alvisi, B. Hoppe, and K. Marzullo. Nonblocking and orphan-free message logging protocols. In *Proceedings of the 23rd Fault Tolerant Computing Symposium*, pages 145–154, June 1993.

[4] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, and causal. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, pages 229–236, June 1995.

[5] L. Alvisi and K. Marzullo. Trade-offs in implementing causal message logging protocols. In *Proceedings of the 15th ACM Annual Symposium on the Priciples of Distributed Computing*, pages 58–67, May 1996.

[6] O. Babaoglu, R. Davoli, and A. Montresor. Group membership and view synchrony in partitionable asynchronous distributed systems: Specifications. Technical Report UBLCS-95-18, Department of Computer Science, University of Bologna, September 1996.

[7] O. Babaoglu and A. Schiper. On group communication in large-scale distributed systems. *ACM Operating Systems Review*, 29(1):62–76, 1995.

[8] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. Optimal primary-backup protocol. In *WDAG 1992*, pages 362–378, 1992.

[9] The CLEO Collaboration. Home page. http://www.lns.cornell.edu/public/CLEO.

[10] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, Carnegie Mellon University School of Computer Science, 1996.

[11] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in Horus. In *Proceedings 15th Symposium on Reliable Distributed Systems*, pages 140–149, October 1996.

[12] A.S. Grimsaw and W.A. Wulf. Legion – a view from 50,000 feet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, August 1996.

[13] S. Landis and S. Maffeis. Building reliable distributed systems with CORBA. *Theory and Practice of Object Systems*, 3(1):31–43, 1997.

[14] K. Marzullo, M. Ogg, A. Ricciardi, A. Amoroso, F.A. Calkins, and E. Rothfus. Nile: Wide-area computing for high-energy physics. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 49–54, September 1996.

[15] R. Moore and R. Klobuchar. Distributed object computation testbed home page. http://www.sdsc.edu/DOCT.

[16] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 56–65, Pozman, Poland, June 1994.

[17] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Replica consistency in CORBA objects in partitionable distributed systems. *Distributed Systems Engineering*, 4(3):139–150, September 1997.

[18] A. Ricciardi, M. Ogg, and K. Marzullo. The Nile system architecture. In *Eleventh International Conference on Systems Engineering*, pages 414–419, July 1996.

[19] A. Ricciardi, O. Ogg, and F. Previato. Experience with replicated distributed objects: The Nile project. Technical Report TR-PDS-1997-012, University of Texas at Austin Department of Electrical and Computer Engineering, December 1997.

[20] L. Sabel and K. Marzullo. Simulated fail-stop in asynchronous distributed systems. In *Proceedings of the Thirteenth Symposium on Reliable Distributed Systems*, pages 138–147, October 1994.