# Six Misconceptions about Reliable Distributed Computing

Werner Vogels, Robbert van Renesse and Ken Birman

*Dept. of Computer Science, Cornell University*†

*{vogels, rvr, ken}@cs.cornell.edu*

**Abstract**

This paper describes how experiences with building industrial strength distributed applications have dramatically changed the assumptions about what tools are needed to build these systems.

## Introduction

For the past decade the reliable distributed systems group at Cornell has been building tools to support fault-tolerant and secure distributed systems [3]. These tools (Isis, Horus and Ensemble) have been successful in an academic sense because they functioned as research vehicles that allowed the community to explore the wide terrain of reliability in distributed systems. They were also successful commercially as they allowed industrial developers to build reliable distributed systems in a variety of industrial settings. Through our interactions with industry we learned a tremendous amount about the ways distributed systems were built and about the settings in which the Cornell tools were employed.

What we found is that professional developers have applied these tools in almost every possible way… except for the ways that we intended. By building major distributed applications such as the New York & Swiss Stock Exchanges, the French Air Traffic Control and CERN's data collection facility the professional development community revealed that tools of the sort we offered are needed and yet that our specific toolset was not well matched with the requirements of serious practical builders [2]. This was not because the technology was somehow “incorrect” or “buggy” but simply because of the particular ways in which distributed systems of a significant size are built. These surprising experiences led us to reevaluate our tools and how we can improve support for building reliable distributed systems.

In this paper we want to discuss the misconceptions we had about how distributed systems are built. We believe that many others from the research community shared our assumptions, and that they still are the basis for new research projects. It is thus important that others learn from our mistakes

## The misconceptions

The design of interfaces, tools and general application support in our systems was based on beliefs concerning the manner in which developers would build distributed applications under ideal circumstances. This methodology presumed that only the lack of proper technology prevented the developer from using, for example, transparent object replication. It is only now that the tools capable of supporting this kind of advanced technology have been built and used by the larger public, that we have come to recognize that some of the technology is just too powerful to be useful. We offered a Formula One racing car to the average driver.

† The research by the Reliable Distributed Computing Group at Cornell University is supported by DARPA/ONR under contract N0014-96-1-10014 and by grants from IBM, Siemens AG, Intel Corporation and Microsoft Corporation

In this section we describe some of the more important idealistic assumptions made in the earlier stages of our effort. In retrospect, we see that our assumptions were sometimes driven by academic enthusiasm for great technology, sometimes overly simplistic or sometimes just plain wrong.

### 1. Transparency is the ultimate goal.

A fundamental assumption of our early work was that the developer seeks fault-tolerance and hopes to achieve this by replacing a critical but failure-prone component of an existing system by a high reliability version, obtained using some form of active replication [1]. Thus although we considered client/server transparency as a potential source for problems, we also believed that replication transparency from the point of the server writer was a laudable goal. Developers should not have to worry about replication strategies, failure conditions, state transfer mechanisms, etc., and be able to concentrate on the job at hand: implementing server functionality.

It turns out that server developers *want* to worry about replica configuration, intervene in failure detection or enabling explicit synchronization between replicas. There was only a small class of server applications where the designer did not care about the impact of replication, and most of these involved server replicas that needed no access to shared resources and were not part of a larger execution chain. The majority of systems in which transparent replication is used became more complex, suffered reduced performance and exhibited potential incorrect behavior traceable to the lack of control over how replication is performed. There is a strong analogy with starting additional threads in a previously single threaded program, where the designer is not aware of the added concurrency.

### 2. Automatic object replication is desirable.

The prevailing thinking over the past decade has been that object systems represent the pinnacle for transparent introduction and presentation of new system properties (components did not exist until recently!). Objects provided an unambiguous encapsulation of state and a limited set of operations. By using language or ORB features we could automate the replication of objects without the need for any changes to the objects, while using state machine replication. Products such as Orbix+Isis [5] and projects such as Electra [6] have been successful in a technical sense in that they succeeded in implementing these techniques, but their practical success in the hands of users was very limited.

In addition to suffering the transparency limitations just described, automatic object replication exposed problems in the area of efficiency. Apart from the fact that the state machine replication was a very heavyweight mechanism when used with more than two replicas, a generic replication mechanism needs to be conservative in its strategies and may be very limited in terms of available optimizations. When allowing the developer control over what and especially when to replicate, optimizations can be made using the semantics of the application.

### 3. All replicas are equal and deterministic.

One of the more surprising violations of our early assumptions was that many services did not behave in a deterministic manner. The systems we were confronted with were often complex multithreaded or multi-process systems, running on several processors concurrently, containing massive subsystems designed and built by different teams. Trying to isolate a generic template for transforming such systems into reliable ones using a toolkit approach turned out to be impossible, because generic solutions invariably depend upon strong assumptions about deterministic behavior.

### 4. Technology comes first.

It makes good sense for an academic research group to focus on a technology for its own sake. After all, academic research is rewarded for innovation and thoroughness. As tool developers, however, we need to focus on the application requirements first and then use technology only if it matches. When an academic group transitions technology into commercial use it suffers from a deep legacy of this early set of objectives. Obviously, there are success stories for academic-to-commercial transition, but they rarely involve general-purpose tools. In particular, academics tend to focus on the hardest 10% of a problem, but tool developers for industrial settings need to focus first on doing an outstanding job on the easiest 90% of the problem, and dealing with the residual 10% of cases only after the majority of cases are convincingly resolved.

### *5. The communication path is most important*

Many of the reliability techniques add extra complexity to the system design and impose performance limitations. We spent years building super-efficient protocols and protocol execution environments, trying to eliminate every bit of overhead, and sometimes achieving truly dazzling performance. Developers, however, are less concerned about state-of-the-art performance: their applications are heavyweight and the processing triggered by the arrival of messages is significant. The parts of the system they stress more and where performance really mattered is that of management. Consistent membership reporting, bounded-delay failure handling, guarantees on reconfiguration and bounds on the costs of using overlapping groups are the issues they truly care about. Moreover, to slash performance we often accepted large footprints, in the form of code bloat from inline expansion of critical functions and increased memory for communication buffering. The user often favors a smaller solution at the cost of lower performance.

### *6. Client management and load balancing can be made generic.*

We taught the user to replicate critical servers, then load balance for high performance, and we argued that it could be done in a generic, very transparent manner. Indeed, this fits the prevailing belief, since many RPC systems work this way (consider cluster-style HTTP servers). But the situation for replicated servers proved much more complex as soon as clients had a longer lasting relation with a server instance, with state maintained at the server. In those cases client management could no longer be made generic, without resulting to protocol specific tricks or using mechanisms that reduce performance significantly. Another aspect was that server developers wanted to have strong control over the distribution of clients among the servers. They almost always were interested in using application-specific knowledge to group client connections together to maximize server-processing efficiency.

## Conclusion

What conclusions can we draw from our experiences? The central theme in most of the erroneous assumptions was that of *transparency*. Trying to achieve transparent insertion of technology, to augment services with reliable operation, seemed a laudable goal. Looking back we have to conclude that transparency has caused more pain and tragedy than it has provided benefits to the programming community. We believe the conclusion is warranted that we need to avoid including strong transparency goals in our future work. Similar conclusions can be drawn from experiences by other groups that are currently evaluating their research results [4].

This conclusion drives a new research agenda for our group. We will develop a new system, at this moment code-named *Quintet*, in which distribution in all its aspects is made explicit. The new system will have many high-level tools available to help the programmer with operations such as replication, but the developer decides, what, where and when to replicate. Some of the management and support tools, such as a shared data structure toolkit, rely on generic replication, but the developers that use them are aware of the implications of importing this functionality [9].

The decision to give full control to the developer is in strong contrast with the current trends in reliable distributed systems research, where transparency is still considered the Holy Grail. The new systems [7.8.10] experience the same limitations that the serious use of our systems exposed. Only by restricting the useable model will these systems be able to support developers in a consistent manner.

Quintet will not restrict the traditional programming model in any way and while providing the developer with more effective tools to do his/her job.

## References.


[1] Birman, K.P., *Building Secure and Reliable Network Applications*. Manning Publishing Company, and Prentice Hall, 1997

[2] Birman, K., "Reliable Multicast Goes Mainstream", Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS) Spring 1998 (Volume 10, Number 1)

[3] Birman, K., and Renesse, R. van, “Software for Reliable Networks”, in *Scientific American*, May, 1996

[4] Guerraoui , R., Felber, P., Garbinato, B., and Mazouni, K., "System Support for Objects Groups" in *Proceedings of ACM Conference on Object Oriented Programming Systems, Languages and Applications* (OOPSLA'98) , October 1998

[5] Landis, S.,and Maffeis, S. "Building reliable distributed systems with CORBA," *Theory and Practice of Object Systems*, John Wiley & Sons, New York, 1997. 5.

[6] Maffeis, S., "Adding group communication and fault-tolerance to CORBA," in *Proc. Usenix Conf. on Object-Oriented Technologies*, June 199

[7] Narasimhan, P., Moser, L.E., and Melliar-Smith, P.M., "Exploiting the Internet Inter-ORB Protocol interface to provide CORBA with fault tolerance," in *Proceeding .of the 3rd Conference. on Object-Oriented Technologies and Systems*, Portland, OR, June 1997.

[8] Singhai, A., Sane, A., and Campbell, R.H., “Quarterware for Middleware”, *Proceedings of the International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, 1998.

[9] Vogels, W., Chipalowsky, K., Dumitriu, D., Panitz, M., Pettis, J., Woodward, J., "Quintet, tools for building reliable distributed components", submitted for publication March 1998

[10] Wang, Y., and Lee, W., "COMERA: COM Extensible Remoting Architecture", *Proceedings of the 4th Conference. on Object-Oriented Technologies and Systems*, Santa Fe, NM, April 1998.