Workshop on Design Principles for Experimental Distributed Systems,
Purdue Univ., Oct. 1986

Experience with Charlotte: Simplicity versus Function in a Distributed Operating System

Raphael A. Finkel, Michael L. Scott, William K. Kalsow,
Yeshayahu Artsy, Hung-Yang Chang, Prasun Dewan, Aaron J. Gordon,
Bryan Rosenburg, Marvin H. Solomon, Cui-Qing Yang

University of Wisconsin — Madison
1210 W. Dayton Street
Madison, WI 53706

Extended abstract

1. Introduction

Charlotte is a distributed operating system currently in production use at the University of Wisconsin–Madison [1, 2, 5]. Charlotte is intended as a testbed for developing techniques and tools for exploiting large-grain parallelism to solve computation-intensive problems. It runs on the Crystal network [4], which contains 20 VAX-11/750 computers interconnected by an 80 Mbps token ring. Charlotte was constructed over the course of approximately 5 years, going through several distinct versions as the underlying hardware and our ideas for implementation changed.

Our starting point was a set of axioms that defined the environment of the project, goals that defined our hopes, and design decisions that helped us to reach those goals. Our implementation taught us that the goals are not easily reached. In particular, the issue of *simplicity* is quite complex; the quest for simplicity in various areas often conflict with each other. The purpose of this paper is to explain the lessons we learned and motivate the steps we took while learning those lessons.

The axioms that constrained Charlotte's design were:

- **Charlotte must run on a multicomputer.** A *multicomputer* is a collection of conventional computers, each with its own memory, connected by a communications device. The tradeoffs between multicomputers and multiprocessors, which share memory, include scalability (multicomputers have a greater potential), grain of parallelism (multicomputers are suited only to large-grain parallelism), and expense (it seems less expensive to build a multicomputer).
- **Charlotte should support a wide variety of application programs.** Since the field of distributed computing is still young, we did not want to limit ourselves to client-server, pipeline, master-slave, or other communication paradigms and algorithm structures.
- **Policies and mechanisms should be clearly separated.** In order to experiment with policies, we did not want to embed them in the core of the operating system. Instead, we decided to place mechanisms in a *kernel* that is replicated on each machine. Policies are governed by *utility processes* whose location is generally irrelevant to the objects that they govern.

Our overall goals were simplicity and function:

- **Charlotte should provide adequate function.** The communication facilities of Charlotte should be appropriate to application programs covering the wide variety of communication paradigms. This function should allow graceful degradation if some machines fail.
- **Charlotte should be simple.** Both the kernel and the utility processes should have this property. Simplicity has many dimensions. We intended Charlotte to be **minimal**, in the sense that it would not provide features that were not needed, and *efficient*, in the sense that the primitives could be executed quickly. We were also concerned that Charlotte be both easily implemented and easily used. We were only partially successful in meeting these latter goals.

Charlotte is unique in the design decisions we made. Our initial design decisions included the following:

- **Processes do not share memory.** This decision allows us to make inter-process communication completely location independent. It mirrors the fact that Charlotte runs on a multicomputer.

* This work was supported in part by NSF grant MCS-8105904, Arpa contract number N0014-82-C-2087, a Bell Telephone Laboratories Doctoral Scholarship, and a Tektronics Doctoral Fellowship.

- **Communication is on reliable, symmetric, bi-directional paths named by capabilities.** Two-way paths are justified below. The use of capabilities (described more fully below) promotes an object-based model for clients and servers. Processes exercise control over who may send them messages. An action by one process cannot damage another, so long as the second takes basic precautions. Capability-based naming also facilitates experimentation in migration for load sharing.
- **Absolute information is stored at each end of a communication path.** The information describes the machine, process, and path number of the other end of the path, and is stored in the kernel. Our choice of absolute information is an implementation decision, and is not necessitated by the communication semantics we chose. The alternative (used in V kernel [3], for example), is to store hints at each end of the path, and to use broadcast as a fallback strategy when the hints fail.

The resulting operating system fulfills our goal of function and simplicity in some ways but not in others. We will show how the design decisions we made to satisfy our axioms and goals incurred implementation cost and required extra levels of software. Although the IPC semantics were intended to be simple, it turned out that supposedly orthogonal features interacted in complex ways and required a complex implementation. Application programmers found that while the IPC semantics make it possible to write highly concurrent programs, they also make it easy to commit subtle programming errors. To reduce the frequency of errors, we designed the LYNX programming language to regularize the use of Charlotte's primitives.

We learned the following important lessons from our experiences.

- Simple primitives interact in complex ways.
- It is not easy to make use of asynchronous primitives.
- Appropriate higher-level tools mitigate the programming problems.
- Absolute distributed information is hard to maintain.
- Message screening belongs in the application layer.
- Middle-level primitives are usually at the wrong level.

On the whole, Charlotte has lived within its constraints and achieved its goals. Charlotte provides a functioning framework for writing distributed applications, and many projects are underway in designing such applications. LYNX has been valuable beyond the multicomputer framework in which it was developed; it has been successfully ported to the Butterfly machine [6]. Given LYNX, library routines, and our connector facility, it is fairly easy to write correct and intelligible applications.

2. References

1. Artsy, Y., H-Y Chang, and R. Finkel, "Charlotte: design and implementation of a distributed kernel," Computer Sciences Technical Report #554, University of Wisconsin–Madison (August 1984).
2. Artsy, Y., H-Y Chang, and R. Finkel, "Interprocess communication in Charlotte," Computer Sciences Technical Report #632, University of Wisconsin–Madison (February 1986).
3. Cheriton, D. R. and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pp. 128-139 (In *ACM Operating Systems Review* 17:5) (10-13 October 1983).
4. DeWitt, D., R. Finkel, and M. Solomon, "The Crystal multicomputer: Design and implementation experience," Technical Report 553 (To appear, IEEE Transactions on Software Engineering) , University of Wisconsin–Madison Computer Sciences (September 1984).
5. Finkel, R. A., A. P. Anantharaman, S. Dasgupta, T. S. Goradia, P. Kaikini, C-P Ng, M. Subbarao, G. A. Venkatesh, S. Verma, and K. A. Vora, "Experience with Crystal, Charlotte, and Lynx," Computer Sciences Technical Report #630, University of Wisconsin–Madison (February 1986).
6. Scott, M. L., "Lynx reference manual," BPR 7, Computer Science Department, University of Rochester (March 1986).