Cerberus - a tool for debugging distributed algorithms

F. Carter and A. Fekete
Department of Computer Science, University of Sydney
Madsen Building F09, University of Sydney 2006, Australia.
email: fekete@cs.su.oz.au

Abstract
Distributed applications are hard to program. They are particularly prone to subtle race conditions, deadlocks, or similar errors in the underlying distributed algorithm. This paper describes a tool which can assist the designer in debugging a distributed algorithm early in the software lifecycle. The tool takes a high-level abstract description of the algorithm, and an even more abstract requirements specification; it simulates an execution until a discrepancy arises between algorithm and specification; it then assist the developer to explore backwards and forwards through the execution till the error is understood.

Keywords
Distributed programs, debugging, simulation, formal methods

1 INTRODUCTION

Programming a distributed application is even more difficult and error-prone than writing other software systems. One source of difficulty is the lack of programmer experience with the languages and tools used in coding: the domain is relatively new, and standards are still developing, so different systems are incompatible and many unnecessary obstacles are placed in the programmers' path. Whole books have been written to assist with these arcane details (Stevens, 1990). Distributed applications however differ from tasks in other domains in that even before the coding phase is reached, the design phase is often affected by fundamental algorithmic errors, which are hard to detect, and are not amenable to being fixed with small corrections. For example, in a sequential program, a common mistake in algorithm design is an "off-by-one" loop, which fails to check the last element of a sequence; this is easily detected by testing with a range of extreme inputs, and is fixed by changing the termination test in the loop. In contrast, a distributed algorithm may have a "race condition", where the error is revealed only when a particular pattern of message delays occurs, and the correction requires a completely new approach. The greater difficulty of design for distributed algorithms is clearly evident in the high rate of errors among papers written by experts and accepted in prestigious journals. As as extreme case, Knapp (1987) discusses the sad sequence of incorrect algorithms for detecting deadlock.
Because a distributed application is so vulnerable to errors in algorithm design, we believe that the development of these applications would benefit from a tool that allows the debugging of the algorithm itself, in a rather abstract early form. The designer should be able to explore the executions of the underlying algorithm intended for their system; only once a sound design is chosen would coding take place. This paper describes a tool of this sort. It is called Cerberus and a first version has been implemented.

It is important for the reader to distinguish the sort of tool we describe, which is used for debugging a distributed algorithm at the design phase, from those which can assist in debugging a deployed distributed system after the coding is completed. The latter sort of tool must itself be a distributed program, collecting information at multiple sites in a network, and attempting to determine whether or not certain global conditions are satisfied. Because remote information is always out-of-date, a debugger for distributed systems is very hard to build. Babaoglu and Marzullo summarise the theory behind these systems in chapter 4 of the book edited by Mullender (1993).

The view of software development in this paper is based on top-down refinement: the designer starts from a requirements specification of the service the system is expected to provide to its clients. As described by Fekete (1993), this service specification will generally be presented as a global, abstract, state transition system. Next the designer decides on a fundamental algorithm that will provide this service. For example, the algorithm might involve a token traversing the network, or it might be based on a replicated state machine (Schneider, 1990). This basic algorithm is described as a collection of separate abstract state transition systems, one (for each site in the network) representing the part of the system at one site, and others (such as buffers) which provide inter-site communication. In Cerberus, both requirements specification and proposed distributed algorithm are presented in a particular syntax which is based on a semantic model called Input-Output Automata (Lynch and Tuttle, 1989), which has been extensively used in research papers for describing distributed algorithms. The Cerberus tool is used to detect errors in the basic algorithm. Once no more errors are discovered, and the designer is confident in the correctness of the algorithm, the individual site components can be further refined to efficient code in a conventional programming language. This requires converting the state-transition description used by Cerberus to flow-of-control in a language like C; one must also replace abstract data structures like sets by efficient implementations.

The top-down style of development supported by Cerberus is in contrast to the more common bottom-up building of distributed applications, where the designer takes individual components that already exist, and combines them in different configurations to meet various goals (perhaps writing additional clients to make calls on the components). This bottom-up style of composition is expected by recent standards such as CORBA or Microsoft's OLE Component Object Model; it is also supported by prototype tools such as the Software Architect's Assistant (Ng et al, 1995).

Non-determinism is a key feature of distributed algorithms, and a central reason for the frequency of major errors in their design. Even though each node in the system is completely predictable in its responses to messages, the whole system has a very large set of executions, each corresponding to a particular pattern of unpredictable message delays. Since the system has no control over these delays, an algorithm is considered correct only if every possible execution produces the desired outcomes. The core of the Cerberus tool is to simulate one execution of the distributed algorithm (if no error is detected in this, another execution is simulated). The tool allows the designer to control the execution