Straddled Clients and Adaptive Remote Procedure Calls

- Abstract -

This paper presents the issues surrounding the evolution of the client-server paradigm, from the simple star network of a mainframe and its dumb terminals, to the modern browser and web server that we find common today. The trend toward mobile use of under-powered devices over noisy wireless links, however, has uncovered a performance imbalance between a client's resources and a client's software resource requirement. While some work has been done to address this imbalance, the focus of client-server performance improvement is most often on the server: The usual conclusion suggests the clients will be happy if we provide infinite resources at the server. We believe there is an upper limit to that solution and present an alternative. We propose a ***straddled client*** *that splits into two equal parts, and sending one to the server to begin ongoing negotiations over which client part uses which portion of the overall client resource requirement. The division of labor between these two parts is adaptive, being a function of the instantaneous resource utilization levels of the client and server systems.*

I. Introduction

The Internet was designed to interconnect powerful computers on a fixed network, but the trend today is toward its mobile, interactive use. Access through laptop and palmtop computers is commonplace, and access to World Wide Web sites through personal digital assistants (PDAs) and sophisticated cellular telephones is on the rise. These devices are actually very small, battery-powered, hand-held, radio-linked computers. Their simple input mechanisms, their narrow, often monochrome displays, limited physical memory, and low-powered processors make them our focus of performance improvement efforts. Providing satisfactory service to an underpowered client device over a noisy, congested wireless channel is one of today's biggest challenges.

II. Background

Dumb terminals first provided users with interactive access to data over a star network, into a centrally located computer. The popularity of on-line database access soon taxed the resources of that central computer and its database management system to their limits. In the mid-1970s, front-end mini-computers were used as concentrators, preprocessing database queries, thereby lifting some of the burden off the central computer [1]. The introduction of smart terminals (and the later use of terminal emulation software) moved the burden of preprocessing a user's query to that user's client terminal. This new client-server paradigm reduced the load of the central processor by making each client responsible for checking its own queries before requesting service.

The underpowered mobile wireless device described above, however, suggests that the resource utilization burden must necessarily shift back again, at least somewhat, toward the higher powered, tethered server.

The popularity of inter-networked computers opened up many new avenues of research. In the early 1980s, Birrel and Nelson developed the Remote Procedure Call paradigm, or RPC [2]. They proposed that a database query could be modelled as a subroutine call, and that a single, common subroutine might be located on a central computer. When the client software needs to send a query, it calls a subroutine to do so. This *client* subroutine is actually a stub that communicates with its *server* counterpart over the network. The server calls the centrally located subroutine that interacts with the database software. Results of the query are returned via the same client-server communication link. Neither the client nor the server software are aware of any network, inter-process communication, or even any necessary data conversion, should the client and server computers represent their data differently internally. This means the software is easier to develop: all of the complexities surrounding the centralization of data are of no concern to the programmer.

Later in the 1980s, the International Standards Organization (ISO) developed their Open Systems Interconnect (OSI) model. They proposed a 7-layer hardware and software architecture over which inter-networked computers might communicate [3]. The ISO model generalized, as well as standardized the roles of each level, pointing out the need for common interface standards between them. So long as the software in the transport layer of an IBM mainframe, for example, followed the same rules as that of its counterpart in a DEC mini-computer, there could be communication and cooperation. The OSI model opened up the possibilities for client-server as well as peer-to-peer relationships between unlike computing environments.

In the late 1980s and early 1990s, there was "…a reaction to the complexity of OSI *application* protocols, such as X.400 and X.500. Rather than standardize a lot of functionality, why not allow clients to introduce the functionality they need?" [4]. The mobile agent was born.

In 1994 the Telescript, and later Java programming languages addressed the issue of how a mobile agent might be written on one kind of machine and then executed on any one of a number of others. The *virtual machine* execution environment described in [5] and [6], and later the Java-enabled browser allowed the client to tap into a server and have immediate access to the interface that allowed them access to that server's data. When a client connects to a server the server sends its interface to execute on the client's computer, eliminating the needs for application level standards beyond the original connection and download strategy.

In the mid-1990s White took a close look at the performance of the RPC client-server paradigm and suggested a variation that he called Remote Programming (RP). Instead of the client sending a message and receiving a response for every action, why not, he asked, send the client to the server along with a list of requests? The client, the agent, uses inter-process communication to send itself across the network to the server system. The agent then uses some local application protocol to access the server data and sends itself back to the client system, with a list of responses [7].

The resources of the client and server hardware must be closely matched to the needs of the client and server software, or a performance imbalance will occur. The system will perform as poorly as its weakest performing component. The solutions to date have focused on the server: providing seemingly infinite server resources should satisfy all the

clients. We see some issues developing around the use of resource-poor clients over a mobile wireless link, suggesting (to us) that an all-powerful server may not be sufficient.

Historically, the trend has been to reduce the load on the server by making the client stronger and smarter, and giving it a bigger share of the load. More recently, the client software has been playing a more interesting role in the relationship. We propose to give it *choice*, to decide (in real-time) how much of the resources it needs should come from the client device and how much will come from the server.

III. The Performance Imbalance

Today's solution to the problem of supporting thin client access to the World Wide Web is to provide a portal, a front-end processor that *clips* HTML data from a web page before presenting it to the user. If the client device need only display "Stock Ticker:" instead of a full-color display and a list of search engine options, the user is provided information in a manner that preserves its resources.

Making these thin clients smarter and more powerful in order to eliminate the front-end processors is not an immediate option. The trend in client device design is to make them even *thinner*! This provides even longer operation times without increasing the size or weight, or the number of batteries. Smaller devices will fit unobtrusively in one's pocket or purse, or on one's wrist [8].

Our proposed solution is to introduce a new computing paradigm called ***straddled clients***, where the client software process creates a second copy of itself on the server, straddling the client-server architecture. These two copies <u>communicate</u> with each other in order to determine the resource utilization load between themselves, on both the client and server systems. They <u>adapt</u> to changes by negotiating the points at which function calls within the client become a remote procedure call: the distribution of resource utilization can be adjusted in real-time.

IV. The Straddled Client

Given the sophistication of a modern client, particularly an Internet browser, its *call tree* may be quite complex. Replicating a browser, leaving one on the client machine and creating an identical copy on the server [9] allows control of the distribution of resources required by the client and server processors. Having an RPC point (or set of points) at one place or another within the call tree might lead to several different load distributions between the client and the server. This allows the load balancing to be adjusted dynamically, adapting to the client and server system load, the RF link characteristics, or both, as necessary.

Nearly every programming language has the concept of a *main* program and support for invoking some sort of *subprogram*. The main program calls the subprogram, it performs some well-defined function, and returns control just after the point of call. Complex programs have numerous subprograms and correspondingly complex call trees, as shown in Figure 1.

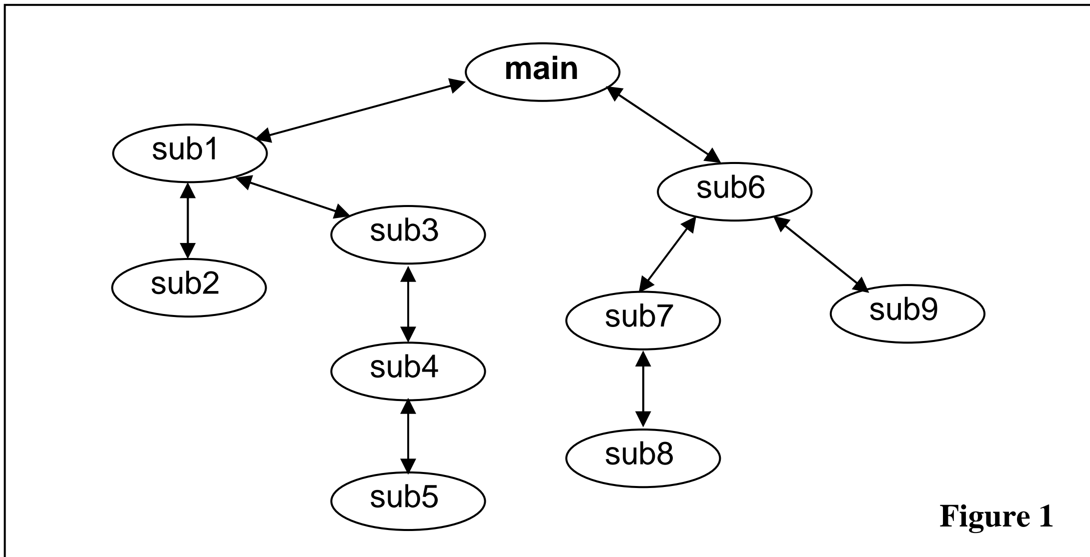
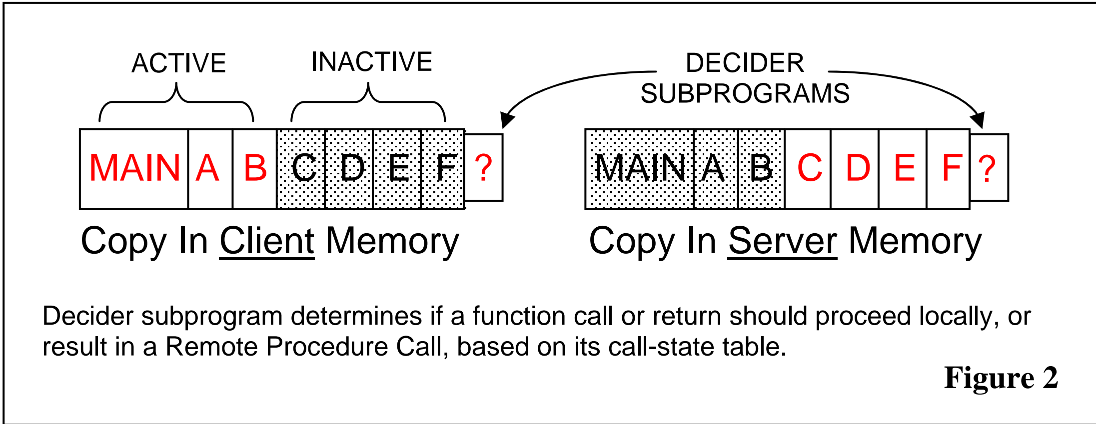


Figure 1

The example has a main program and nine subprograms. For discussion purposes, we assume each one contributes exactly 10% of the overall resource utilization load, and the execution time is uniformly distributed over all of them. If the main program executes only on the client device and the subprograms execute on the server, the distribution of resource utilization is 10-90. Ten percent of the resources required by the client process will be used on the client device, and ninety percent will be used on the server. If the main program and four of its subprograms are active on the client device, and the other five are active on the server, then the distribution is 50-50. Utilization of virtual memory is the same on both machines because a complete copy of the client process exists in both the client device and the server machine's memory space. Statically allocated variables and dynamically acquired buffers are a topic requiring further investigation.



Decider subprogram determines if a function call or return should proceed locally, or result in a Remote Procedure Call, based on its call-state table.

Figure 2

Calling points and subprogram entry points may be examined at compile time by a pre-compiler, which creates a table that may be accessed at run time. Each direct call can be replaced with a call to a central *decider* subprogram, where the point of call, the subprogram called, and all the passing parameters are known (Figure 2). Known too, is the current state of these points of call. When any function is called, the *decider* subprogram decides whether to continue with the call to the local copy, or to marshal the calling parameters and proceed with the RPC, according to the dynamic call-state table. Any changes to the call-state table must be reflected on both sides of the client-server

link, requiring a replication control mechanism for this table, a distributed shared data object.

V. Conclusion:

Possible call states, along with their respective utilization percentage breakpoints can be used to dynamically adjust the loading on the client and server, based on statistics gathered in real time, and passed between decider subprograms. The decision to change the call states can be negotiated between them by piggybacking control messages onto the parameter packets required by RPC.

Further research into the complexity of handling dynamically allocated buffer space, and statically allocated variables, seems indicated. There may be considerable overhead incurred by giving decider subprograms control at every call point, and there may be unexpected difficulties in maintaining their call-state table data. An adaptive approach to performance improvement has provided considerable return on investment in the past, however, suggesting the development of an instrumented prototype as a next step.

VI. References

1 Tsichritzis, D. C. and Lochovsky, F. H., *Data Base Management Systems*, New York, NY: Academic Press, 1977.

2 Birrell, A. D. and Nelson, B. J., "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems,* vol. 2, pp. 39-59, February 1984.

3 Day, J. D. and Zimmerman, H., "The OSI Reference Model," *Proc. of the IEEE*, vol. 71, pp. 1334-1340, December 1983.

4 White, J. E., "Telescript Retrospective," Mobility: Processes, Computers, and Agents, pp. 493, ACM Press, 1999.

5 Lindholm, T. and Yellin, F., *The Java Virtual Machine Specification*, 2nd Edition, Palo Alto, CA: Sun Microsystems, imprinted by Addison-Wesley, 1999.

6 Venners, B., *Inside the Java Virtual Machine*, New York, NY: McGraw-Hill, 1998.

7 White, J. E., "Mobile Agents," Mobility: Processes, Computers, and Agents, pp. 461-492, ACM Press, 1999.

8 Briody, D. and Latti, M., “Wireless Web whirlwind,” January 25, 2000, at http://www.cnn.com/2000/TECH/computing/01/25/wireless.web.idg/index.html

9 Vrenios, A. "Process Cloning in C/Linux," *Linux Gazette*, On-line Magazine, March 2000, Issue #51, http://www.linuxgazette.com