

# Replicated Clients and Adaptive RPC

## - Research Project Summary -

Prepared for the

Consortium for Embedded and Internetworking Technologies

### Overview:

The focus of this project is to investigate the effectiveness of an Application Program Interface (API) as a low resource utilization (lightweight) replacement for an ORB-based system, a reduced-cost-and-complexity approach to high availability support on a multicomputer architecture. The API is designed for ease of use and to ensure the continued functioning of a distributed application in the event of equipment down time, due to scheduled maintenance or unanticipated failures.

Our test bed software has three major components: an input load generator, the API, and an application. The API is the focus of our investigation; the application merely provides an operational framework. All software will be written in the C programming language and will run on a cluster of PCs interconnected by an Ethernet network. The API will be instrumented for performance measurement and tested under a variety of loading and failure conditions.

### Approach:

Remote Procedure Call (RPC) emulates a local procedure call by packing all the passing parameters into an interprocess communication packet and sending it to a remote processor, where server-side software makes a local call to the target function. In this manner, processes may be split into common client-side software packages running in parallel, using RPC to call a server-side function. Many clients have simultaneous access to a single, centrally located service.

The application and API will run on a network of eight, nearly identical PCs. API performance will be monitored and displayed in near real-time by a ninth PC. A tenth PC will provide the input transaction load and will accept and record any responses, displaying response time statistics from the external client's perspective. The transaction load generator will allow interactive adjustments to the packet size distribution, the packet arrival rate distribution, and their interrelationship, the number of client sessions that the packet stream represents.

Generally, we view the ORB-based approach to distributed system implementation as a centralized solution to a distributed problem, with an unacceptably high impact to overall system performance. Our solution will be to take a long-overdue look at the architectures, implementing distributed algorithm alternatives where applicable, in an attempt to develop a lightweight replacement. We believe that we can build an API that will facilitate distributed software development without ultimately limiting its practicality.

Distributed systems rely on message passing: one process passes information on to another, compelling it to take some action, or not, based on the message content. Ever since Leslie Lamport's seminal work on distributed system clocks in 1978, information has been piggybacked onto these messages to disseminate local state information. We intend to develop a similar concept that will present accurate remote system loading to each local node so that when each thread needs to select a remote node to call, it has a high probability of finding it up and ready to offer the service of its resources.

One byproduct of our work would be a more efficient use of network bandwidth. When the number of packets dropped or reacquired due to processing overhead is reduced, throughput, in terms of overall data rates or the number of simultaneous connections, is increased without improvements to the existing network communications hardware. This alone might be of enormous consequence to the future design of distributed systems.

## **Results:**

We effectively separated the user's application from our API, creating an interface that allows easy conversion of an existing single system, concurrent transaction server to one on a multicomputer. Support for distributed services are embedded in our API, and may be of no concern to the user, who needs only declare his existing functions to the API through a series of function calls during initialization. This is a quick operation, follows the structure of the user's original server software design, and has no impact on performance during subsequent operation.

Our prototype software architecture performed poorly at first. We instrumented the software, compiled and analyzed execution phase time averages, optimized certain sections of the software, and validated its improved scalability. We enhanced the design of the API toward a generalized RPC paradigm, where any remote process could call from, or return to, any other, employing global resource utilization and load balancing algorithms.

Time ran out before we could complete the implementation, however, leaving us with a working prototype, a near real-time performance monitor, and an enhanced distributed software design. Global resource utilization data proved ineffective as a load balancing mechanism, multiple requests arriving at the same remote node caused lost packets, suggesting a requirement for a global locking mechanism, and the high availability additions to our system were never implemented.

Our next step would be to improve local representation of available remote system resources and to add a global locking mechanism. Finally, we would examine the system for vulnerabilities, adding software and hardware fault tolerance solutions where applicable.