

# Secure Dynamic Reconfiguration of Scalable CORBA Systems with Mobile Agents<sup>\*</sup>

Fabio Kon Binny Gill Manish Anand Roy Campbell M. Dennis Mickunas

Department of Computer Science  
University of Illinois at Urbana-Champaign  
<http://choices.cs.uiuc.edu/2k>

**Abstract.** As various Internet services, e-commerce, and information systems permeate our lives, their continual availability becomes a dominant issue. But continuing software evolution requires system reconfiguration. Running systems must upgrade their components or change their configuration parameters. In addition, Internet services often need to serve thousands or millions of users. This scenario raises three conflicting issues: availability, configurability, and scalability.

We propose the use of mobile reconfiguration agents for the efficient, secure, and scalable dynamic reconfiguration of Internet systems. We describe a CORBA object-oriented framework that supports dynamic reconfiguration and allows customization to different kinds of computing environments ranging from PDAs and embedded systems with limited resources to powerful workstations.

## 1 Introduction

The scope of Internet services continues to expand, stretching to new fields and encompassing more and more human activities in the virtual world of the web. Activities such as management of bank accounts, reading news, accessing medical information, filing income taxes, submitting articles to scientific conferences, getting weather and natural disaster information, and consolidating votes in elections or referenda stress the importance of service availability, reliability, and security. However, the rapid evolution of software requires frequent code updates and reconfiguration. Maintaining the flexibility and rapid growth of these systems while ensuring the service requirements on such a scale as the Internet is a challenging problem.

A flexible service can adapt to meet changing requirements and resource availability. However, flexibility usually conflicts with availability. A service provider must often shut down, reconfigure, and restart a service in order to update or reconfigure it. But, disruption may result in business loss, as in the case of e-commerce, or it may put lives in danger, as in the case of systems delivering disaster information. Research in dynamic configuration [11] seeks solutions to

---

<sup>\*</sup> This research is supported by the National Science Foundation, grants 98-70736 and EIA99-72884EQ. Fabio Kon is supported in part by CAPES-Brazil, proc. 1405/95-2.

this problem. By breaking a complex system into smaller components and by allowing the dynamic replacement and reconfiguration of individual components with minimal disruption of system execution, it is possible to combine high degrees of flexibility and availability.

Scalability also conflicts with flexibility and it is usually not addressed by research in dynamic configuration. Services such as video on demand, on-line bookstores, and search engines must maintain a high level of parallelism in order to fulfill their demand. World-wide services with high-bandwidth requirements such as on-line sales of software must be readily available in different countries. The problem is that reconfiguring and updating a distributed collection of servers is troublesome. It may lead to wasted bandwidth – as duplicated information is sent through the same Internet lines – and it introduces several problems related to consistency and fault-tolerance. In this paper, we present an architecture that solves the problems of availability, flexibility, and scalability in an integrated and secure way.

## 2 Application Scenarios

The application of mobile agents to a variety of distributed computations can improve performance significantly. It also provides a uniform way to deal with code mobility and disconnected operations. We now look at two different uses of mobile agents for dynamic configuration.

**Multimedia Distribution System.** We first identified the need for mobile reconfiguration agents for Internet systems when we developed a component-based system for scalable distribution of multimedia streams [6]. This system realizes distribution using a network of *Reflectors* spread across a wide area. Each Reflector works as a relay, receiving input multimedia packets from a list of trusted sources and forwarding these packets to other Reflectors or to end-user clients. The communication between each pair of reflectors can use a variety of protocols depending upon the situation.

Our experience with this system ranges from point-to-point audio conferencing to live broadcast of video and audio through a network of more than 30 Reflectors delivering multimedia streams to millions of users across five continents [6]. The difficulty in carrying out those experiments, managing more than 30 Reflectors in a wide-area network, exposed the necessity of flexible mechanisms for runtime reconfiguration as well as agent-based tools for optimizing bandwidth utilization in code distribution and component reconfiguration.

Using the Reflector system, we carried out the live broadcast of the NASA Mars Pathfinder mission from a PC station at the Jet Propulsion Laboratory, a broadcast that lasted for several months. During that period, we found programming errors in the Reflector code and were forced to update the executable code in all the machines manually. In addition, system reconfigurations, like increasing the maximum number of allowed clients or adding new multimedia channels, were also cumbersome since the administrator had to connect to each of the Reflectors separately.

In the following sections we present our solution to the problems mentioned above. We reorganized the Reflector code to permit the dynamic replacement of some of its fundamental components. Thus, using the reconfiguration agent infrastructure described in sections 3 and 4, Reflector administrators are now able to distribute new implementations of Reflector components and to replace old implementations by new ones on-the-fly.

**Mobile Computing and Factory Control Systems.** Mobile reconfiguration agents also aid in industrial settings where embedded software systems control factory machinery. The administrator can walk on the factory floor, inspecting the physical machinery while using a PDA to inspect the state of the digital control devices by contacting them through infra red or wireless connections. Using a graphical management tool, the administrator may create agents that visit the controllers in that section and return the collected information to the PDA. Using this information, the administrator can then create appropriate agents and inject them into the network, reconfiguring the system to optimize production.

### 3 Configurable Middleware

To support dynamic (re)configuration of distributed component-based systems, we built *dynamicTAO* [7], a CORBA ORB that enables on-the-fly reconfiguration of its internal engine. *dynamicTAO* exports an interface for loading and unloading components into the ORB runtime and for inspecting the ORB configuration state. It can also be used for dynamic reconfiguration of applications running on top of the ORB and even for the reconfiguration of non-CORBA applications.

The *dynamicTAO* architectural framework is depicted in figure 1. The *Persistent Repository* stores component implementations in the local file system organizing them in hierarchical *categories*. It offers methods for manipulating categories and the implementations of each category. Once a component implementation is stored in the local repository, it can be dynamically loaded into the system runtime. A *Network Broker* receives reconfiguration requests and forwards them to the *Dynamic Service Configurator*, which supplies common operations for dynamic configuration of components at runtime.

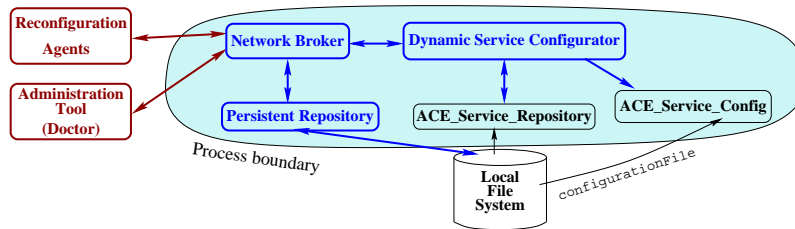


Fig. 1. *dynamicTAO* components

```

interface DynamicConfigurator
{
    typedef sequence<string> sList;
    typedef sequence<octet> implCode;

    sList list_categories ();
    sList list_impls (in string categName);
    sList list_loaded_impls ();
    sList list_domain_components ();
    sList list_hooks (in string compName);
    string get_impl_info (in string implName);
    string get_comp_info (in string compName);
    string get_hooked_comp (in string compName,
                           in string hookName);
    string get_latest_vers (in string categName);

    long load_impl (in string categName,
                  in string implName,
                  in string params)

    void hook_impl (in string loadedImpl,
                  in string compName,
                  in string hookName);

    void suspend_impl (in string loadedImpl);
    void resume_impl (in string loadedImpl);
    void remove_impl (in string loadedImpl);
    void configure_impl (in string loadedImpl,
                       in string message);

    void upload_impl (in string categName,
                    in string implName,
                    in implCode binCode);
    void download_impl (in string categName,
                      inout string implName,
                      out implCode binCode);
    void delete_impl (in string categName,
                    in string implName);
};

```

Fig. 2. The *DynamicConfigurator* interface

We delegate some of the basic configuration tasks to components of the ACE framework [5] such as the *ACE\_Service\_Config* – used to process startup configuration files and manage dynamic linking – and the *ACE\_Service\_Repository* – used to manage loaded implementations.

System administrators or “intelligent” reconfiguration software can drive the reconfiguration of applications running on top of *dynamicTAO* by connecting to a network broker using a given TCP/IP port or a CORBA IDL interface. This scheme uses the Distributed Configuration Protocol (DCP) (see <http://choices.cs.uiuc.edu/2k/DCP>), which defines commands for code distribution, inspection of dynamic state, reconfiguration of runtime software architecture, and reconfiguration of running components. Figure 2 shows the *DynamicConfigurator* IDL interface that supports DCP.

The first nine operations shown in figure 2 are used to inspect the dynamic structure of a particular domain and to retrieve information about the different abstractions. In our model, a *category* represents the type of a component and each category typically contains different *implementations*, i.e., dynamically loadable code stored in the local persistent repository. Once an implementation is loaded into the system runtime, it becomes a *loaded implementation* and can be associated with a logical *component* in the application domain. Finally, components have *hooks* which represent inter-component dependencies, reflecting the runtime software architecture of the application and its underlying ORB.

The *load\_impl* operation dynamically loads a component implementation from the ORB persistent repository into the system runtime and starts running it. *hook\_impl* attaches it to a hook in one of the components in the domain. *remove\_impl* finalizes and unloads a component from the runtime and *configure\_impl* is used to send component-specific configuration messages to a given component. *upload\_impl* allows a remote entity to store code for an implementation in the persistent repository, so that it can later be linked to a

running process. `download_impl` allows a remote client to retrieve the code for an implementation from the ORB persistent repository.

## 4 *dynamicTAO* Reconfiguration Agents

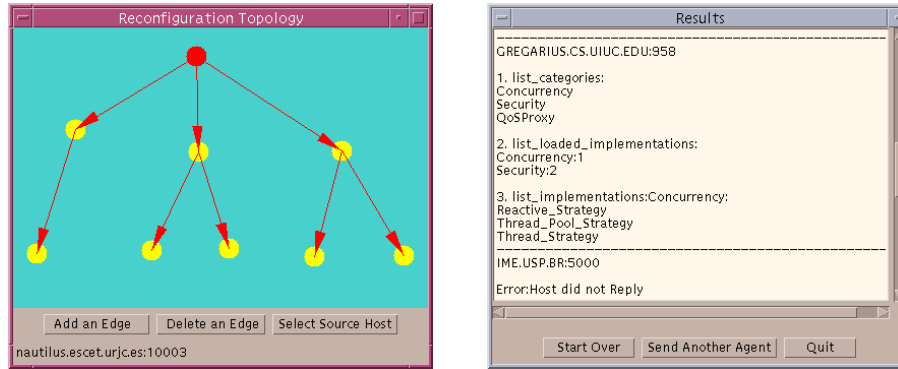
Our initial version of *dynamicTAO* and its DCP implementation required a point-to-point connection between the administrative tool and an application process in order to configure that particular application process. But this approach did not scale. If a system administrator needed to upgrade a certain component of an on-line service composed of 30 replicas, it was necessary to connect to each replica separately, upload the new implementation of the component, and reconfigure the replica.

As a first solution to this problem we considered implementing a management front-end that would allow administrators to type sequences of reconfiguration actions that would be sent to a list of application nodes. Although this approach would improve scalability by simplifying the work of the administrator, it would not solve the problem of bandwidth waste, i.e., sending large amounts of duplicated information across long-distance Internet lines.

The solution we adopted was to allow administrators to organize the nodes of their Internet systems in a hierarchical manner for reconfiguration purposes. The administrator specifies the topology of the distributed service as a directed graph and creates a *mobile reconfiguration agent* which is injected into the network. The reconfiguration agent then visits the nodes of this graph of interconnected ORBs. Each ORB first replicates and forwards the agent to neighboring nodes, then executes the agent locally, processing its reconfiguration commands, and, finally, collects the reconfiguration results, sending them back to the neighboring agent source.

Using this approach, the administrator can organize the reconfiguration hierarchy to optimize the data flow between distant application nodes. The reconfiguration commands are executed in parallel in the various nodes, improving response time. If desired, the graph may contain different levels of redundancy so that the system can tolerate the failure of some of the nodes in the reconfiguration network. To enhance scalability and decentralize administration, different administrative domains may select a *Domain Representative* for the purpose of receiving reconfiguration agents. The global administrator only needs to include the *Domain Representative* in the distribution graph. It is then the responsibility of the representative to forward the agent to the relevant nodes in its domain.

In this model, the agent behavior is defined by its reconfiguration script and graph, which are specified by the administrator. In Section 4.3, we describe how we extend our model to support *autonomous* agents that decide by themselves which nodes to visit and which reconfigurations to perform. Overall, our goal was to develop a flexible infrastructure that could support different flavors of reconfiguration agents.



(a) Creating the graph edges

(b) Results returned by agent

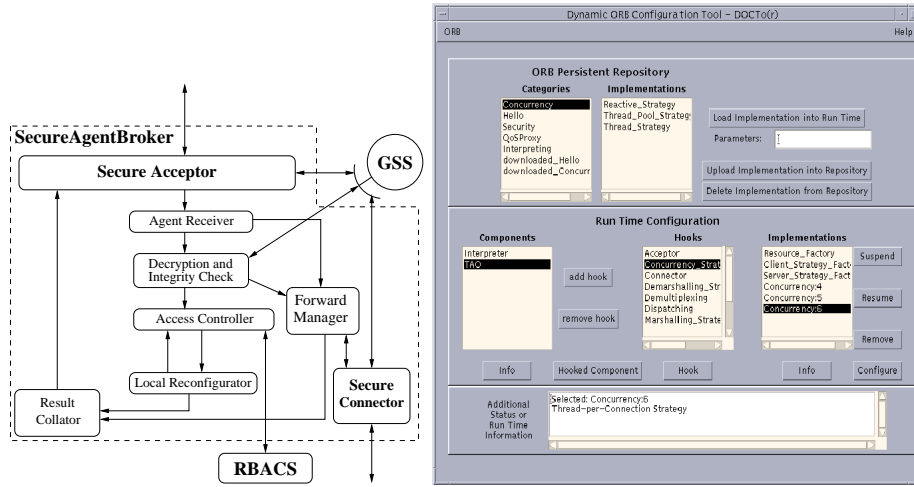
**Fig. 3.**

#### 4.1 Implementation

Our implementation of the mobile reconfiguration agent infrastructure consists of (1) extensions to the original implementation of *dynamicTAO* for receiving, processing, and forwarding reconfiguration agents, and (2) a Java graphical front-end for specifying reconfiguration graphs and for assembling and sending reconfiguration agents. The administrator selects those ORBs that will be part of the reconfiguration graph and, using the mouse, draws directed edges connecting the graph nodes as shown in figure 3(a). Once the reconfiguration graph is established, the graphical front-end assists the administrator in building a script of DCP commands that are codified into a reconfiguration agent. Finally, the administrator instructs the tool to send the agent to the initial node in the graph.

Each agent contains (1) a copy of the reconfiguration graph (so that different agents can operate on different graphs at the same time), (2) a script of DCP commands to be interpreted on each node in the graph, and (3) a unique sequence number which is used to avoid processing duplicated copies of the same agent on the same node. When the agent comes back to the administrator node, the results returned by all the nodes in the reconfiguration graph are finally displayed to the administrator through the graphical front-end. The system also draws attention to the nodes that did not reply within a given timeout. Figure 3(b) shows the results returned by an agent containing three DCP inspection commands (also called an *inspection agent*).

Our agent dissemination system can be seen as a simple, lightweight implementation of reliable multicast over TCP/IP. We intentionally adopted this simple and reliable solution for agent transmission because it lets us concentrate on the dynamic configuration aspects of the problem. However, depending upon the topology of the reconfiguration network (for example, if the nodes have many output edges), it may be worthwhile to use other underlying protocols such as IP-Multicast for distributing agents. In that case, however, the complexity of the



(a) The Secure Agent Broker (b) The Doctor configuration tool  
 Fig. 4.

middleware would be much higher since it would have to guarantee reliability over a non-reliable protocol.

#### 4.2 Security

To provide security, our architecture supports access control, authentication, and encryption by using our group's CORBA implementation of the standard General Security Services (GSS) API [8]. We adopted the Role-Based Access Control model (RBAC) because it is more flexible and easier to manage than the traditional DAC approach used in systems like Unix and Windows [13].

We extended *dynamicTAO* to include an instance of a *SecureAgentBroker*, a subclass of the *Network Broker* that supports secure reconfiguration agents. As shown in figure 4(a), it uses a component implementing the *GSS API* and a *Role-Based Access Control Service (RBACS)*. The administrator who wishes to inject a reconfiguration agent into the network establishes a secure connection with the root node in the distribution graph by using a GSS-enhanced version of the administrative front-end. We use the GSS encryption mechanisms for data privacy and integrity. Thus, unauthorized principals are not able to decrypt the agent or tamper with it. The system ensures authenticity by using credentials and delegated credentials supplied by the GSS. The RBACS provides mechanisms by which the *SecureAgentBroker* enforces access control. Thus, the system is able to recognize multiple roles such as, for example, *admin*, *user*, and *alien*. The *admin* might have all the rights for reconfiguration and inspection; the *user*, only inspection; and the *alien* might have no rights.

Whenever a *SecureAgentBroker* needs to forward a reconfiguration agent to another node, it first establishes a secure connection with it. Then, it forwards the encrypted agent along with the credential and role name of the original author of the agent. On receiving a reconfiguration agent, the broker decrypts it and checks for integrity and authenticity by using the GSS API. It then forwards the agent to other nodes as dictated by the distribution tree accompanying the agent. The *Access Controller* checks the authorization for each DCP command the agent issues by contacting the RBACS. If the access is valid, the broker executes the agent locally and merges the results with the results from all its children nodes before forwarding the collated results to its parent.

### 4.3 Plugging Different Interpreters - Java Agents

Our default implementation supports agents with simple reconfiguration scripts based on the DCP protocol. On the one hand, this allowed us to write an extremely lightweight interpreter that processes the agents efficiently. This is very appropriate for PDAs and embedded systems that can be found, for example, in the factory control scenario described in section 2. On the other hand, it limits the expressiveness and autonomy of the reconfiguration agents. By using a more powerful language for reconfiguration such as Java, the administrator is able to write more sophisticated agents that can use the agent and ORB state to make decisions about their actions. To achieve that, we organized our infrastructure as an object-oriented framework to which different kinds of agent interpreters and mechanisms can be plugged. For example, by plugging a Java Virtual Machine, we added support for Java agents.

We encapsulated Sun's JVM into a component that can be dynamically loaded into the *dynamicTAO* domain; this component exports an interface that contains operations for loading and interpreting Java bytecode. In addition to the reconfiguration topology graph, that now can be modified on-the-fly, the Java agent carries its dynamic state from node to node and it contains Java bytecode that is interpreted inside a sand-box so that its actions can be limited. By using the *DynamicConfigurator* IDL interface (see figure 2), the Java bytecode can issue inspection and reconfiguration commands to the ORB domain. Based on the result of these commands the agent can (1) update the state it carries from node to node, (2) change its behavior to adapt to the environment and user preferences on each node, and (3) modify the reconfiguration graph, deciding to migrate to a different set of nodes. In that manner, our framework offers both simple agents based on DCP scripts for environments with limited resources and powerful Java agents that can carry state and modify its behavior along the way.

### 4.4 A Discussion on Fault-Tolerance and Consistency

Our experience with distributed computer systems has shown that several applications do not require strict consistency between the components in the different nodes. When deploying the multimedia distribution system described in section 2

for example, we were able to use different versions of Reflector components simultaneously. In many cases, even if a component update was successful in part of the nodes and failed in another part, the distributed system could continue working gracefully until all the updates were achieved manually.

To keep the system small and efficient, we opted not to provide strong guarantees about the execution of the reconfiguration commands in the current implementation. On a single node, some reconfiguration commands may fail while others succeed; it is the responsibility of the agent to catch the exceptions raised by the failed commands and treat them accordingly. Also, reconfiguration commands may succeed on some nodes and fail on others. To cope with these errors, the system detects the failures and the graphical front-end (shown in figure 3(b)) displays to the administrator where they occurred and the respective diagnosis messages. Then, using a separate tool called *Doctor* (the Dynamic ORB Configuration Tool in figure 4(b)), the administrator can connect to the node where the failure occurred, inspect its state and reconfigure the node manually. By navigating through ORB and application components interactively, the administrator can investigate the causes of the errors, and fix them.

On the other hand, it is possible to achieve the effects of atomic transactions using the functionality we are offering through the Java agents. On an individual node, the Java agent can be programmed to store the previous state of the system before starting to make any changes. Should an error occur during the reconfiguration process on that node, the Java code can catch the exceptions and bring the configuration back to the initial state.

To achieve the effects of atomic transactions on the distributed system as a whole, one can use a different technique. According to our experience, reconfiguration failures usually happen when a new component is loaded (and the dynamic loader cannot link it) or when the new component is initialized (and it fails to locate the functions or allocate the resources it needs). Thus, we instruct the administrator to separate its reconfiguration in two phases resembling the two-phase commit protocol for atomic transactions [3]. First, the administrator sends an agent to load all the required components and to initialize them by executing the *load\_impl* operation (and, if needed, *configure\_impl*). If an error occurs, the administrator chooses between sending a new agent to unload the new components from all the nodes or using *Doctor* to fix the individual errors. If the component implementations are loaded and initialized successfully in all the nodes, the administrator sends a new agent to attach them to the proper applications and ORB components, which would correspond to the *commit* phase of the two-phase commit protocol.

We found this to be sufficient for most of the application scenarios with which we work. However, some other applications require more strict guarantees of the ACID properties [3] for atomic transactions. For example, if we want to reconfigure the marshalling and unmarshalling components in all the ORBs in a distributed system, it is important to guarantee an *all-or-nothing* property in the transactions. If an ORB replaces its marshalling mechanism and one of the ORBs to which it sends requests does not change its unmarshalling mechanism, they

might no longer be able to communicate. To support that, one could extend our infrastructure by using standard transaction mechanisms for distributed systems [2] and recent protocols for mobile agent fault-tolerance [14].

## 5 Experimental Results

We have used our infrastructure in a variety of systems including Solaris 7, IRIX 6.5, Linux Red Hat 6.1, and Windows NT and 98. To evaluate the response time and relative performance gains made possible by our infrastructure, we established an intercontinental testbed with the collaboration of researchers in Brazil and Spain. The testbed consisted of three groups of machines: two Sun Ultra-60 and one Ultra-5 machines running Solaris 7 at `cs.uiuc.edu`, three 333MHz PCs running Linux RedHat 6.1 at `escet.urjc.es`, and three 300MHz PCs running Linux RedHat 6.1 at `ic.unicamp.br`.

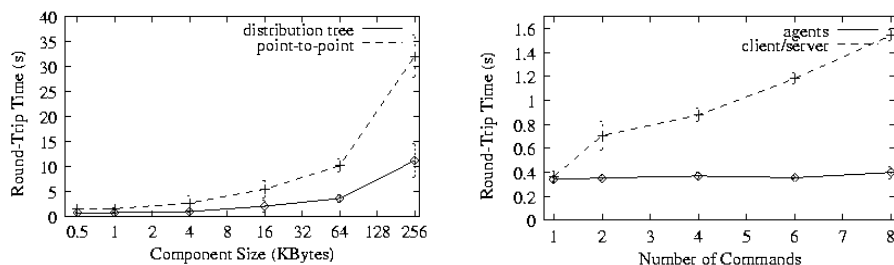
The machines inside each group were connected by 100Mbps Fast Ethernet while the groups were connected among themselves through the public Internet. We executed several instances of a test application running on top of *dynamic-TAO* and injected different kinds of agents in this network.

To avoid drastic oscillations in the available Internet bandwidth and latency, and to minimize undesired interference, we carried out the experiments during the night<sup>1</sup>. The average bandwidth and round-trip latency between our lab at `cs.uiuc.edu` and `escet.urjc.es` were 76KBps and 170ms, respectively. Between our lab and `ic.unicamp.br`, 32 KBps and 270ms, respectively.

In our first set of experiments, we measured the total round-trip time for executing five inspection commands (*list\_categories*, *list\_impl*, *list\_loaded\_impl*, *list\_domain\_components*, and *get\_comp\_info*) using different techniques. By sending the five commands to a single remote node using an agent, we obtained a 68% performance improvement compared to sending a separate request for each command without using agents (362 ms against 1128 ms). When sending agents to the nine nodes using the distribution tree shown in figure 3(a), we obtained a 60% performance improvement compared to a point-to-point approach (847 ms against 2127 ms).

To measure how the benefits of using a distribution tree varies with the size of the agent, we sent a series of agents carrying the code for components of different sizes to be installed in the remote nodes. As shown in figure 5(a), as the size of the component being uploaded increases, the relative gain of using a distribution tree instead of point-to-point connections increases significantly. This happens because the agent “multicast” mechanism minimizes the use of the public Internet, which is the bottleneck in this experiment. In the graphs, each value is the arithmetic mean of 10 runs of each experiment and the vertical bars represent the standard deviation. Finally, figure 5(b) shows the comparative performance between the agent approach and conventional client/server requests as the number of requests increases from one to eight. The client/server times

<sup>1</sup> When we ran the experiments during times of high network traffic and congestion, the performance numbers were even more in favor of our mobile agents approach.



(a) Uploading a component to 9 nodes

(b) Agents vs. client/server

Fig. 5.

were measured by building an application that sends the commands contained in the agent as separate requests transmitted sequentially via TCP/IP connections. Since the overhead of processing this kind of agent (around  $1ms$ ) is negligible compared to the latency of long distance Internet lines, the performance of both approaches when sending a single command is roughly the same. As we increase the number of commands in each experiment, the total completion time for the agent barely varies while the completion time for the client/server application increases rapidly as expected.

Although our implementation could still be improved significantly with more tuning and optimizations, these preliminary results are very encouraging. They demonstrate clearly that mobile agents can provide extreme performance improvements for the reconfiguration of wide-area distributed systems.

## 6 Related Work

Previous and ongoing research in dynamic configuration [11] use architectural description languages, connectors, and dataflow models to represent the structure of complex applications and, in some cases, use configuration languages to specify dynamic modifications in this structure.

This paper shows how we combine two apparently unrelated research areas (mobile agents and dynamic configuration) to manage the dynamic reconfiguration of distributed applications in an efficient, scalable, and secure way.

Ranganathan *et al.* developed a middleware for reconfigurable distributed scripting [12] that also involves agents and reconfiguration. However, while we use mobile agents to reconfigure scalable component-based applications, their work focuses on the dynamic reconfiguration of agent-based applications that use their middleware. Thus, our motivation and goals are different.

One of the most common concerns related to mobile agents is security. Requirements include the authentication of the involved parties and the protection of the execution environments from malicious agents. Albeit, the most challenging issue, the protection of the mobile agents from the execution environ-

ments, is still mostly unresolved. Methods based on provably-secure languages and proof-carrying code [9] have been proposed to provide protection for execution environments. JavaSeal [16] shows how to isolate agents from one another. Protecting the agent from malicious hosts is, in general, more difficult and can be partially solved by approaches like the time-limited black-box security and clueless agents [15].

Our approach is to provide secure communication channels for the agents to traverse, protecting them from external attacks. The hosts use Role-Based Access Control [13] to limit what the agents can do.

Baldi and Picco presented an elaborated quantitative model for mobile code performance [1]. Puliafito, Riccobene, and Scarpa carried out an analytical comparison among the client-sever, remote evaluation, and mobile agents paradigms [10]. Their results help evaluating under which circumstances the use of mobile agents is beneficial.

Ismail and Hagimont carried out an empirical performance evaluation of their infrastructure for Java mobile agents, comparing them to Aglets [4] and to approaches based on RMI. Their experiments focused on single-hop agents on *intra*-continental networks. Our experiments involved single- and multi-hop agents on *inter*continental networks. Both works demonstrated that the mobile agent model can lead to significant performance improvements.

## 7 Conclusions

As Internet services become pervasive in our society, we become more and more dependent on their availability. However, software is in continuous evolution, which requires running systems to be updated and reconfigured on-the-fly with minimal disruption.

This paper proposes the use of mobile reconfiguration agents for secure, efficient, and scalable dynamic reconfiguration of Internet systems. We described a prototype implementation of such a mobile reconfiguration agent infrastructure based on a CORBA-compliant ORB supporting dynamic reconfiguration. Our framework can be customized either with efficient, lightweight agent interpreters – for environments with limited resources such as PDAs and embedded systems – or with more powerful interpreters such as the Java Virtual Machine.

Empirical results show that agents can improve the performance of distributed, dynamic reconfiguration greatly. We believe that, within a few years, the use of mobile agents for management and configuration of large-scale Internet systems will be a common practice.

### Availability

The source code and documentation for *dynamicTAO*, DCP, and the mobile-agent-based reconfiguration engine is available at <http://choices.cs.uiuc.edu/2k/dynamicTAO>.

## Acknowledgments

The authors gratefully acknowledge Luiz Magalhães, Ramesh Chandra, Balaji Srinivasan, Arun Viswanathan, and Vanish Talwar for early work on the reconfiguration agent infrastructure. We thank Francisco Ballesteros and Alexandre Oliva for giving access to their laboratories so that we could perform the wide-area experiments. Finally, we thank Geoff Arnold and the anonymous reviewers for their precious comments on the preliminary versions of this paper.

## References

1. M. Baldi and G. Picco. Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications. In *Proc. 20th Int. Conf. Software Engineering*, pages 146–155, April 1998.
2. J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors. *CAMELOT and AVALON: A Distributed Transaction Facility*. Morgan Kaufmann Pub., 1991.
3. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California, 1993.
4. L. Ismail and D. Hagimont. A Performance Evaluation of the Mobile Agent Paradigm. In *Proceedings of OOPSLA '99*, pages 306–313, Denver, November 1999.
5. P. Jain and D. C. Schmidt. Dynamically Configuring Communication Services with the Service Configuration Pattern. *C++ Report*, 9(6), June 1997.
6. F. Kon, R. Campbell, et al. A Component-Based Architecture for Scalable Distributed Multimedia. In *Proc. 14th ICAST*, pages 121–135, Naperville, April 1998.
7. F. Kon, M. Román, et al. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings Middleware '2000*, number 1795 in LNCS, pages 121–143, New York, April 2000.
8. J. Linn. The Generic Security Service Application Program Interface (GSS API). Technical Report Internet RFC 2078, Network Working Group, January 1997.
9. G. C. Necula and P. Lee. Safe, Untrusted Agents Using Proof-carrying Code. In G. Vigna, editor, *Mobile Agents and Security*, LNCS 1419, pages 61–91. 1998.
10. A. Puliafito, S. Riccobene, and M. Scarpa. An Analytical Comparison of the Client-Sever, Remote Evaluation, and Mobile Agents Paradigms. In *Proc. ASA/MA '99*, pages 278–292, October 1999.
11. J. Purtilo, R. Cole, and R. Schlichting, editors. *Fourth International Conference on Configurable Distributed Systems*. IEEE, May 1998.
12. M. Ranganathan, V. Schall, V. Galtier, and D. Montgomery. Mobile Streams: A Middleware for Reconfigurable Distributed Scripting. In *Proc. ASA/MA '99*, pages 162–175, October 1999.
13. R. S. Sandhu and E. J. Coyne and H. L. Feinstein and C. E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, February 1996.
14. F. A. Silva and R. Popescu-Zeletin. An Approach for Providing Mobile Agent Fault Tolerance. In *Proc. Second Int. Workshop on Mobile Agents*, pages 14–25, September 1998.
15. G. Vigna, editor. *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
16. J. Vitek and C. Bryce. The JavaSeal Mobile Agent Kernel. In *Proc. ASA/MA '99*, pages 103–116, October 1999.