3-5-1996

# An RPC Mechanism for Transportable Agents

Saurab Nog
*Dartmouth College*

Sumit Chawla
*Dartmouth College*

David Kotz
*Dartmouth College*

# An RPC Mechanism for Transportable Agents

Saurab Nog Sumit Chawla David Kotz

Department of Computer Science
Dartmouth College
Hanover, NH 03755

{saurab, chawla, dfk}@cs.dartmouth.edu

Dartmouth Technical Report PCS-TR96-280

March 5, 1996

**Abstract**

Transportable agents are autonomous programs that migrate from machine to machine, performing complex processing at each step to satisfy client requests. As part of their duties agents often need to communicate with other agents. We propose to use remote procedure call (RPC) along with a flexible interface definition language (IDL), to add structure to inter-agent communication. The real power of our Agent RPC comes from a client-server binding mechanism based on flexible IDL matching and from support for multiple simultaneous bindings. Our agents are programmed in Agent Tcl; we describe how the Tcl implementation made RPC particularly easy to implement. Finally, although our RPC is designed for Agent Tcl programs, the concepts would also work for standard Tcl programs.

**Keywords** - Transportable Agents, Remote Procedure Call (RPC), Interprocess Communication.



This work was funded in part by Office of Naval Research (ONR) contract number N00014-95-1-1204. Their support is gratefully acknowledged.

# 1 Introduction

A *transportable agent* is a named program that can migrate from machine to machine in a heterogeneous network. Such programs are a powerful tool for implementing *information agents* because the electronic resources in a user's information space are often distributed across a network and can contain tremendous quantities of data. An agent can move the search engine to the data, reducing network traffic. Agents choose when and where to migrate. They can suspend their execution at an arbitrary point, transport to another machine and resume execution on the new machine. By migrating to the network location of the resource, the program does not need to bring intermediate data across the network and can access the resource efficiently even if the resource developer provides only simple primitives. Thus transportable agents are more efficient than the traditional client-server paradigm and allow rapid development of distributed applications.

Agents often need to communicate, as would be the case between a client and server agent. This communication needs structure, as do all interfaces, but needs flexibility to allow agents to speak on different levels. We developed a remote procedure call (RPC) [BN84] mechanism that, together with our *Agent Interface Definition Language* (AIDL), provides this structure and flexibility for agents.

Our system is built on top of Agent Tcl, a transportable-agent system under development at Dartmouth College [Gra95a, Gra95b]. Transportable agents in this system are written in an extended version of the Tool Command Language (Tcl) [Ous94]. As a result, most of our RPC ideas apply to plain Tcl programs as well.

Figure 1 illustrates the architecture of the system. The topographical position of each component corresponds to its position in the system hierarchy. To write client and server agents, we start by writing an interface in the *AIDL*, which we then present to the *stub compilers*; the stub compilers generate the procedure definitions that are used by the client and server to communicate. Servers register the services they provide with the *nameserver*, which the clients consult. This structure is no different from traditional RPC systems. Our system is different in other ways:

- A new, flexible interface definition language, AIDL, allows default parameters and position-independent parameters.
- Client-server bindings are based on interface matching rather than on names, and there is support for multiple simultaneous bindings.
- *Capabilities* [Fab74] are supported at bind time. A server can accept or reject a bind request based on the validity of the capability provided by the client.

In the ensuing sections we describe the components illustrated in Figure 1 in detail, ending with a performance analysis of our system and a discussion of extensions to the system. We start by describing some related work.

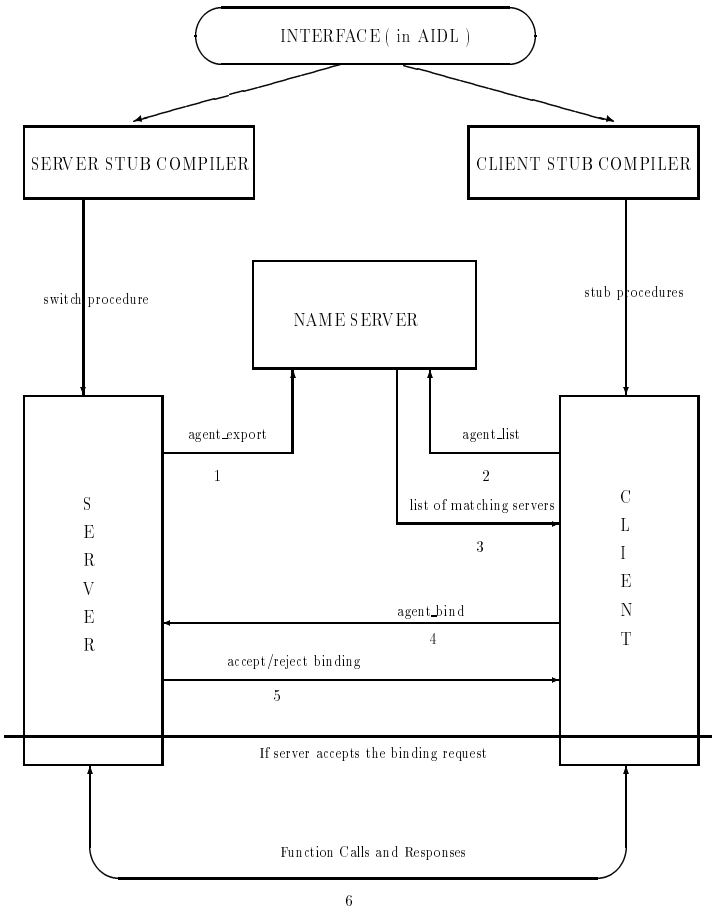


**Figure 1**: The system architecture

## 2 Related Work

Our work blends ideas from two areas, agents and RPC, into a single system. In this section we describe related work in these areas.

The advantages of transportable agents have led to a flurry of implementation work. Three notable systems are Tacoma [JvRS95], Telescript [Whi94], and Agent Tcl [Gra95a, Gra95b]. Tacoma agents are written in Tcl/Horus, which is a version of Tcl that uses Horus [vRHB94] to provide group communication and fault tolerance.

Tacoma agents communicate using the `meet` operation; data to be exchanged is carried in a `briefcase`, which is a collection of folders containing units of data accessible by agents. Tacoma claims to support "RPC style" communication, but this appears to mean that the communication is synchronous. Data, like in the `meet` operation, is exchanged through a

`briefcase`; there is no procedure call.

Telescript is an object-oriented language in which migration is viewed as a basic operation. Telescript agents communicate by obtaining references to each other's objects if they are on the same machine and sending objects to each other if they are on a different machine.

Agent Tcl agents are written in a version of Tcl. Agent Tcl agents communicate using the `meet` operation. This operation establishes a socket connection between the communicating agents. Agent Tcl is free[1] and has been ported to several platforms.

There are many RPC-related systems. Several kinds of nameservers and stub compilers have been written, each with an intended application. Most research, however, is aimed at optimizing RPC speed, through kernel hacks [BALL90] or through the use of specification files to provide information that can be used for optimization [FHL95, AR94]. Our focus is in providing flexibility in the interface specification. The closest work we know is the "Tcl-DP Name Server" [LSR95], which augments the RPC mechanism of Tcl-DP [SRY93], a distributed programming extension to Tcl, with an easy and reliable lookup service. Services are organized in a hierarchical manner as if they were files in a directory. Aliases can be created for services shared by different applications. To get a list of services, say with name "sysdaemon", one can ask the name server to list service names of the form "sysdaemon/*". The name server then returns a list of all matches. Although the list of commands serviced by the name server is similar to ours, they use no IDL for specifying and matching interfaces. Also, clients and servers are processes, not agents.

The Agent RPC system also shares common goals with distributed component technologies like Network OLE [Bro95] and CORBA [COR91]. Both systems try to help clients locate appropriate servers by using interface matching. While the main goals of component-ware are access and location transparency, however, the Agent RPC system is aimed at constantly moving clients (agents) and focuses more on flexibility in interface specification and matching.

# 3 Agent Interface Definition Language (AIDL)

An interface definition serves many important purposes in a client-server environment. A server uses its interface definition to describe its functionality to the world. A server's services may improve over time, however, and it may want to upgrade its interface while still maintaining compatibility with older clients. Since in our system a client uses its interface definition to search for an appropriate server, i.e., one with a matching interface, the aim here is to allow for substantial flexibility in this matching process. The overall goal of the system is to let the client and server evolve independently and still maintain their compatibility.

Agents specify the functional interface they support or want using our Agent Interface

[1] http://www.cs.dartmouth.edu/~agent/

Definition Language (AIDL). AIDL has a flexible format, adding many new features like position-independent parameters and default values that are unavailable in traditional IDLs. The aim is to support extensibility and flexible matching.

An interface is a Tcl list. The elements of this Tcl list are AIDL constant and procedure definitions, which are lists in themselves. Valid interfaces in AIDL can be expressed by the following set of translations.

- interface → constant_list procedure_list
- constant_list → **constant** { constant_definitions } | $\epsilon$
- constant_definitions → { constant_name constant_value } constant_definitions | $\epsilon$
- procedure_list → { procedure_definition } procedure_list | $\epsilon$
- procedure_definition → procedure_name { parameter_list } return_list
- parameter_list → { parameter_name parameter_type } parameter_list |
  { parameter_name parameter_type default_value } parameter_list | $\epsilon$
- return_list → { returnvalue_name returnvalue_type } | $\epsilon$

A constant definition consists of pairs of names and values. The constants thus declared are used mainly for classification and identification purposes. For example, constants allow multiple servers exporting the same functionality to distinguish themselves. It also facilitates simultaneous existence of multiple versions of an interface.

A procedure definition consists of:

- Procedure name.
- List of the parameters (name, type and default value, if any) for the procedure. This list may be empty.
- Return-value information (name, type). This part may also be empty, in which case both the name and type are assumed to be empty.

### An example

A possible travel-agent interface in AIDL might look something like the following example. This example gives a flavor of AIDL and also depicts usage of default values (`credit_card` for the parameter `payment_form`).

```
{constant {version 1} {service travel_agent} {category airlines}}

{get_flight_information {{Origin CityCode} {Destination CityCode}}
```

```
    {flights flight_list}}

{buy_ticket {{payment_form sale_type credit_card} {flight flight_number}}
    {success boolean}}

{get_address_of_travelagent {} {address string}}
```

The constants convey the version number for this server, its service type and its specialization (`airlines`). The names of these constants are not significant to the stub compilers or the nameserver, but are meaningful by convention between clients and servers. The rest of the declarations announce procedures supported by this server.

AIDL procedure declarations have two major features that support its goal of flexibility and extensibility. They are:

- Position-independent parameters.
- Parameters with default values.

These features tremendously ease the restriction of using the same interface for the client and server. These features belong to the interface and not to the individual calls.

In AIDL, parameter declaration order is immaterial for interface-matching purposes. Hence, two procedures that are identical (function name, return value information, and parameter information) except for the order of parameter declaration will be treated as the same during matching. Thus `buy_ticket` could also be declared as

```
{buy_ticket { {flight flight_number} {payment_form sale_type credit_card}}
    {success boolean}}
```

This feature goes a long way towards realizing the goal of being able to find similar services without forcing too much structure on them.

Position-independent default values is another interesting capability of AIDL. Tcl and C++ allow default values for procedure parameters in a restricted manner. Such procedure parameters have to be last in the procedure declaration and cannot be skipped over in case there are multiple default parameters. So if we have the following procedure in Tcl with two formal parameters, each with an associated default value,

```
proc sample { {formal1 default1 } {formal2 default2} }
```

and we call it with one actual parameter, Tcl assumes that the value applies to the first formal parameter (`formal1`).

The separation of interface definition (in AIDL) from implementation (in Tcl) allows the Agent RPC system to present a much more flexible interface to the same procedure. In AIDL, parameters with default values may be intermixed freely with those without (i.e., without any restrictions on the order in which these parameters appear in the AIDL definition). While this definitely helps clients and servers with slightly different interfaces to bind, it also augments the way clients can call this function. If a server defines a procedure `sample` as above, clients may choose an interface for `sample` with no parameters, just `formal1`, just `formal2`, or both `formal1` and `formal2` specified.

AIDL features like position-independent parameters and default values are actually handled by the client and server stubs, as explained in the next section. Figure 2 gives a high-level overview of the underlying process.

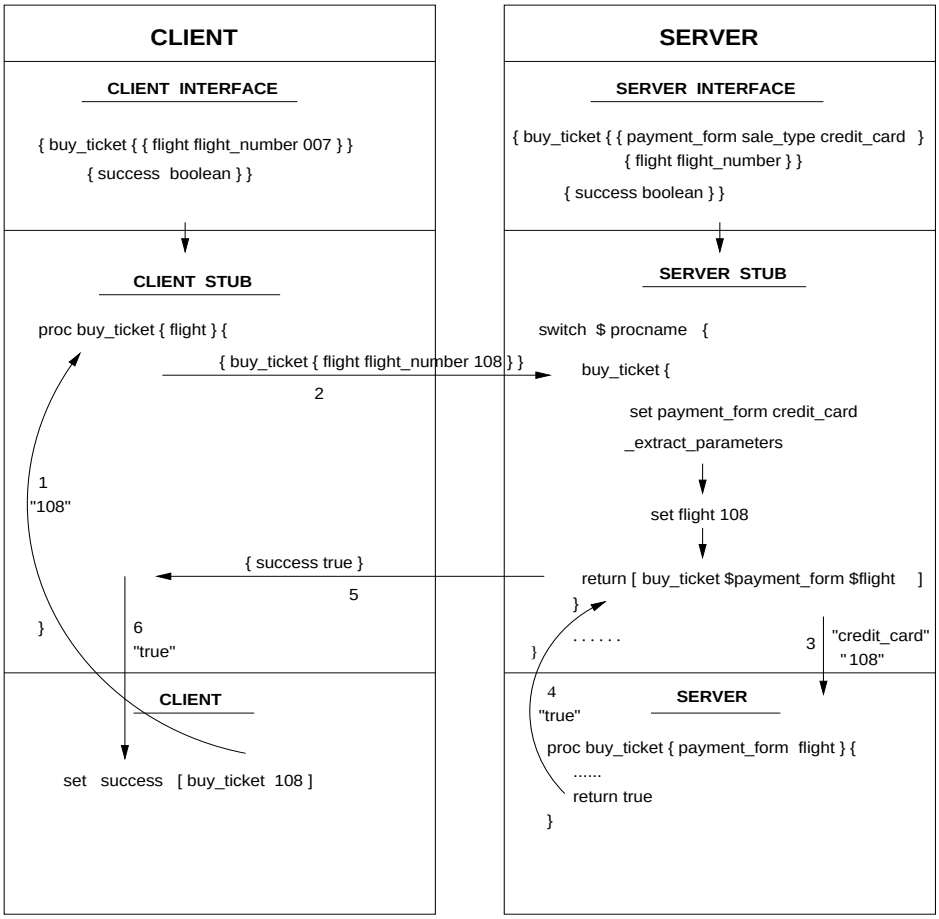


**Figure 2**: The calling mechanism

The client and server have slightly different versions of the `buy_ticket` procedure. However the two match because

- The server has a default value (`credit_card`) associated with the parameter (`payment_form`) missing from the client's declaration.
- The default value of 007 for the parameter flight is ignored by the client stub compiler.

The client stub compiler thus generates a stub procedure with the one parameter (`flight`) while the server stub compiler generates a switch procedure with a call to the server's `buy_ticket` procedure having both the parameters ( `payment_form` and `flight`) mentioned in its interface. The switch procedure fills in the default values into appropriate variables before calling the actual procedure. Thus the server's implementation of `buy_ticket` doesn't know if the client specified a parameter or it was inserted by the server stub.

The result of the actual call is forwarded to the client stub which returns it as the result of the function call.

# 4 Stub Generation and Interface Matching

An AIDL compiler is needed to convert interface definitions into appropriate Tcl code. The compiler performs error checking like detecting duplicate constant, function, or parameter names, incorrect AIDL syntax, and so forth. The Tcl code generated depends on the role of the agent in the system (client or server). Our AIDL compiler is written in Tcl; since AIDL uses Tcl's list syntax, parsing and manipulating AIDL descriptors was substantially easier than most compilers.

Another important function of the AIDL compiler is to generate a compact representation of the compiled interface that can be used efficiently for matching purposes. When the travel AIDL is compiled on the client side, the following compact representation results:

```
{client_stub_compilerv1.0 {constant {category airlines} {service travel_agent}
{version 1}} {{ buy_ticket {success boolean} {flight flight_number} {payment_form
sale_type} } { close_connection {void void} } { get_address_of_travelagent {address
string} } { get_flight_information {flights flight_list} {Destination CityCode}
{Origin CityCode} }}
```

The first part of this long string (`client_stub_compilerv1.0`) is the client compiler's signature. This part is followed by constant and procedure declarations that have been slightly modified:

- The constants have been sorted by their names.
- The procedures have been sorted by their names.
- Parameters for each procedure have been sorted by their names.

- All default-value declarations have been removed from parameter lists.

A server compilation of the same AIDL description would produce the same compact representation except that the signature would be `server_stub_compilerv1.0` and the default value would be present.

The Agent RPC system matches clients with servers based on their interfaces and not the name of the service they seek (as in traditional RPC systems). Sorting helps the interface-matching task performed by the nameserver. The order in which procedure parameters were declared thus no longer effects interface matching. This independence from order of declaration provides Agent RPC with one of its most powerful features: matching based on name and type.

One way to transparently extend a server's interface is to use default values for new parameters. For clients the benefit of having default values lies in the interface-matching process, done by the nameserver. The nameserver will successfully match a client interface against a server interface, if the server interface supports all of the client's procedures. The rule for procedure matching is that, for all procedures mentioned in the client interface, the parameters on both the client and the server must be exactly similar (the order of declaration does not matter) except for those for which the server specifies defaults. So the default parameters act as "don't cares" for the matching process. A client AIDL may or may not have those parameters and that does not affect matching.

Another way to extend a server's interface is to add new procedures. Only clients who request those procedures in their interface will have access to them. In any case, clients using an interface subset can still match.

Our system associates a programmer-supplied identifier with each interface available to an agent (client or server) to help differentiate multiple interfaces and switch between them for purposes of querying and binding. In our implementation, the identifier is actually promoted to an array variable with three fields. One of the fields is the `name` and it stores the compact representation of the interface generated by the AIDL compiler.

## 4.1 Client Compilation

To preserve the abstraction of local function call for RPC, the AIDL client compiler generates stub procedures for the client to use. The stub procedures hide details about inter-agent communication and parameter passing from the agent programmer. Client stubs provide the following functionality:

- Control transfer.
- Packing of input parameters and unpacking return value, respectively. Since all parameters, being Tcl values, are strings, packing and unpacking are much easier than in most RPC systems.

- Sending packed input parameters to the server and receiving results from it. The result is returned to the calling procedure after unpacking.
- Timeout on how long it takes for the server to respond. The timeout limit can be changed when the interface is being compiled.
- Sequenced number generation to detect and remove stale responses.

### 4.2 Server Compilation

Server compilation of interfaces is similar to client compilation except for a few differences:

- Default values for parameters are not ignored. They become a part of the declaration and are used for generating code and the compact interface representation.
- The result of the compilation process is not a list of stub procedures but a `switch` statement. The `switch` statement calls the appropriate routine defined in the server that implements the functionality the client is seeking. The actual server routine does not know anything about defaults either. Default values are applied inside the switch statement, before calling the routine.

The switch statement is enclosed in a switch procedure. For the travel agent interface, the switch procedure looks like

```
proc _switch_proc {procname {param_list {}}} {
     switch $procname {
        buy_ticket {

      # Set the default parameter
             set payment_form credit_card

             # Get the value of parameters from the list specified
             # by the client. This may lead to the overwriting of
             # a default parameter, but that's fine.

             _extract_parameters $param_list

             # call the actual procedure and
             # return the result

             set returnval [buy_ticket ${payment_form} ${flight}]
             return $returnval
        }
```

```
    ... other procedures ...
    }
}
```

Thus when a client calls the `buy_ticket` remote procedure, a variable `payment_form` is created and assigned the default associated value of `credit_card`. `_extract_parameters` is a procedure that assigns variables the values the client specified. So if the client supplied a value for `payment_form`, it will overwrite the default value. The actual server function `buy_ticket` is then called with all the parameters.

# 5 NameServer

The nameserver is an agent with functionality similar to that of the Grapevine system [BLNS82]. It is a database that the servers use to register services and the clients use to locate services. It is responsible for:

- *Registering services*: Servers wishing to export services call `agent_export`, which transmits their `agent_id` (a unique identifier assigned to an agent by the system) and the compiled AIDL `name` to the nameserver. The latter appends the AIDL to its list of existing services. Currently, a server can export only one interface.
- *Matching clients with servers*: Clients wishing to locate servers matching their own compiled interface also use the nameserver. The nameserver serves as a deterministic finite automaton (DFA), matching the specified interface against the list of exported interfaces, and returning the list of matched interfaces. The lexicographic ordering of the components of the interface results in a simple and fast matching process. Strings are matched from left to right, with default parameters being skipped when corresponding parameter name and type pairs do not match. No backtracking is required. The string and list handling features of Tcl make this process quite simple. The `agent_list` command is used by a client to obtain this information.
- *Deleting services*: The nameserver removes the interface from the list upon a recall request from the server. The `agent_recall` command is used by a server to revoke access to the services it provides.

The commands used to initiate conversation with the nameserver are described in Table 1.

# 6 RPC Mechanism

Remote Procedure Call (RPC) is an involved protocol. It requires at least the following steps.

| Category | Command | Description |
|---|---|---|
| Client | `agent_list` | returns a list of names and addresses of servers providing the desired service |
| | `agent_bind` | creates bindings and switches between current bindings |
| | `agent_unbind` | destroys a current binding |
| Server | `agent_export` | inserts the server interface into the nameserver's database |
| | `agent_recall` | deletes the server's interface entry from the nameserver's database |

**Table 1**: Support functions for clients and servers

- Keeping track of services available at various servers.
- Matching a client's needs to an appropriate server.
- Setting up a connection between the client and server(s) (which may include authentication).
- Sending a request from client to server and waiting for the response as well as packing and unpacking the parameters and results.
- Disconnecting the established connection(s) cleanly.

The first two functions are provided by the nameserver, which exists in every agent system along with the agent server. The remaining three are client-server functions and both the client and the server have additional support functions that help them with these stages.

### 6.1 Client Side

The client side of RPC is a bit more involved than the server, because it is the client who initiates conversation. Our agent system allows an agent to have multiple interfaces and each interface to have multiple concurrent bindings. The multiple interfaces are distinguished using programmer-defined array names while the multiple bindings for an interface are cached in the `bind` variable of its respective array. A global descriptor is used to identify the context (interface and binding) in which an RPC call is made.

To prevent the client from getting stuck in an infinite wait, a timeout mechanism is built into the client stubs. If the server does not respond to a client RPC request within a specific period, the client stub throws an exception and returns. The timeout parameter is configurable at AIDL compile time. Timeouts introduce the problem of servers responding after the client has timed out. Such stale responses are weeded out by attaching a sequence number to each RPC request. The sequence number is incremented at each request (successful or not) and is not reused. When a client stub sees a server response with a sequence number not equal to its present one, it discards it as stale.

Support functions for the RPC client are listed in Table 1 and described below

**agent_list** A client's first step is to ask the nameserver for the names and addresses of servers providing the desired service. The client calls `agent_list`, which in turn uses low-level primitives to

- request a meeting with the nameserver (which has a well known name),
- send a message consisting of the `list` command, its own id, and the compiled version of its AIDL interface (obtained from the `name` field of the interface array produced by the stub compiler), and
- wait for the nameserver to do the matching and return a list of server IDs that successfully match the clients interface; this list is stored in the `list` field of the interface array.

**agent_bind** The client's next step is to *bind* to one or more servers. Since multiple interfaces and multiple bindings are permitted, we also need a mechanism to switch between them. `agent_bind` creates bindings and switches between current bindings, as follows:

- The user calls `agent_bind` with the ID of the server. `agent_bind` verifies that the server is in the list returned by `agent_list`. This check prevents the user from trying to bind with arbitrary servers and causing system crashes. Thus `agent_bind` forces the client to call `agent_list` first, to ensure smooth operation.
- If the server ID is valid, the list of currently active bindings is searched. If an active binding already exists, its identifier is made the global context identifier and `agent_bind` returns.
- If no active binding exists, `agent_bind` asks to meet (`agent_meet`) the target server. When the meeting request is granted, a TCP/IP connection is established between the client and the server.
- Once a connection is established, the client sends a capability to the server. This step allows servers to use a capability for authenticating the client's general access rights. In the server stub the capability is passed to a user-defined function hook that returns access permissions (granted or denied).
- After the client's capability has been screened, the server stub responds to the client with an "Access Granted" or "Access Denied" response. If access is denied, both the client and the server close their TCP/IP sockets and `agent_bind` returns an error. Otherwise, the client adds the $\prec server, descriptor \succ$ pair to its list of currently active bindings, sets the global context identifier to the new descriptor, and returns successfully.

| Data Size (bytes) | RPC Time | | Client Stub Time | Server Stub Time | Comm. Time | | | | Local Call Time |
|---|---|---|---|---|---|---|---|---|---|
| | Same Machine | Different Machine | | | Same Machine | | Different Machine | | |
| 0 | 7.903 | 7.023 | 2.511 | 3.210 | 2.182 | 27.61% | 1.302 | 18.54% | 0.101 |
| 1 | 10.082 | 9.213 | 2.633 | 4.867 | 2.582 | 25.61% | 1.713 | 18.59% | 0.117 |
| 10 | 10.207 | 9.365 | 2.702 | 4.914 | 2.602 | 25.46% | 1.749 | 18.68% | 0.186 |
| 100 | 10.764 | 9.882 | 2.754 | 5.291 | 2719 | 25.26% | 1.837 | 18.59% | 0.209 |
| 1000 | 15.104 | 14.658 | 3.233 | 8.901 | 2.970 | 19.66% | 2.524 | 17.22% | 0.443 |

**Table 2**: Agent RPC performance. All times are in milliseconds, and are averages of over 10,000 trials. The data bytes were passed as a single parameter.

**agent_unbind** `agent_unbind` destroys a current binding. It does so by sending an unbind request through the open port to the server and then disconnecting.

### 6.2 Server Side

The server registers its services with the nameserver, and then waits to service incoming requests. Requests could be for:

- *Binding*: a new client wishes to bind to the server. The capability parameter is extracted and passed to a server authentication function. The result of this function determines whether to accept or reject the binding. The client is notified accordingly.
- *Service*: a client with a successful bind wishes to seek a service. The incoming stream is unpacked into the sequence number, the procedure name, and the parameter list, and the switch procedure is invoked. The result obtained from this call is then packed with the sequence number and transmitted back to the client. Note that the sequencing is done on the client side; the server merely sends back the number it received.

The server uses two support functions.

**agent_export** A server registers the services it provides by invoking `agent_export` with parameters `agent_id` and the compiled interface. The result is the insertion of the interface in the nameserver's database.

**agent_recall** A server revokes access to its services by invoking `agent_recall`, with parameter `agent_id`. The nameserver deletes the corresponding entry from its database.

# 7 Performance

We measured the performance of the Agent RPC system using 100 MHz Pentium PCs connected by a 10 Mbit/sec Ethernet. The PCs ran FreeBSD version 2.1 and Agent Tcl version 1.2 (which extends Tcl version 7.5). We measured the end-to-end wall clock time for a complete RPC call, transferring data from client to server, with an empty server procedure and no return data. This experiment was repeated for various parameter sizes, and for both local and remote server agents. The average timing results are presented in Table 2.

The last column of the table shows the time it took to make a local procedure call (same program) with the same amount of data. Since Tcl is inherently slow, this measure is a good benchmark for evaluating the RPC results. In all cases we can see that an RPC took somewhere between 50 to 150 times the time it took for a local call. This ratio is fairly common in most RPC systems.

Note also that communication was a small fraction (17 to 28%) of the total time in all cases (columns 7 and 9). Agent communication was not particularly fast because it uses TCP/IP sockets, but it is not the bottleneck because Tcl interpretation is very slow.

It is interesting that the time for a same-machine RPC was higher than a cross-machine RPC. We believe that this unintuitive result is due to the context switch between the client and server processes on the same machine, which was more costly than the network transfer time.

# 8 Future Work

Having built this system, we now intend to use it for several applications, such as information-retrieval systems, electronic commerce, and workflow.

In addition to this, we are looking at extending the system in the following ways:

- Type checking: currently the type fields of the parameters are ignored. Type checking will become important if we are communicating with agents in other languages, a feature that will be present in a future version of Agent Tcl.
- Security: provide the option to encrypt messages between clients and servers.
- Nameserver: provide the nameserver the ability to update its list of active servers.
- Portals [McK95] [SP95]: we would like to make the nameserver a portal process and use it to build an agent file system.

# 9 Conclusions

We have presented an RPC mechanism for Tcl agents that provides a flexible and easy method of communicating with other agents. We have designed a new IDL for this purpose that allows clients to bind to services that match their specification. The IDL is both simple and flexible, allowing position-independent and default parameters. Tcl's flexibility was instrumental in easing the rapid construction of Agent Tcl and our RPC facility. Our performance results indicate that our system performs reasonably well.

### Acknowledgements

Many thanks to Bob Gray for providing help with the Agent Tcl system and for reviewing this paper. We also appreciate his help in writing parts of the introduction and related work sections.

# References

[AR94] J. S. Auerbach and J. R. Russell. The Concert signature representation: IDL as an intermediate language. In *Proceedings of the Workshop on Interface Definition Languages*, pages 1–12, 1994.

[BALL90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[BLNS82] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Remote Execution. *Communications ACM*, 25(4):260–274, April 1982.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Bro95] Kraig Brockschmidt. *Inside OLE, 2nd Edition.* ISBN 1-55615-843-2. Microsoft Press, 1995.

[COR91] The common object request broker: Architecture and specification. OMG TC Document Number 91.12.1, Revision 1.1, December 1991.

[Fab74] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[FHL95] B. Ford, M. Hibler, and J. Lepreau. Using annotated interface definitions to optimize RPC. Technical Report UUCS-95-014, Utah, March 1995.

[Gra95a] Robert S. Gray. Agent Tcl: A transportable agent system. In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, Baltimore, Maryland, December 1995.

[Gra95b] Robert S. Gray. Transportable agents. Technical Report PCS-TR95-261, Dept. of Computer Science, Dartmouth College, 1995. Thesis proposal.

[JvRS95] D. Johansen, R. van Renesse, and F.B. Schneider. An introduction to the TACOMA distributed system version 1.0. Technical Report 95-23, University of Tromsø, June 1995.

[LSR95] P. T. Liu, B. Smith, and L. Rowe. Tcl-DP Name Server. In *Proceedings of the Tcl/Tk Workshop*, pages 1–13, July 1995.

[McK95] Marshall Kirk McKusick. The virtual filesystem interface in 4.4BSD. *Computing Systems*, 8(1):3–25, Winter 1995.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, Massachusetts, 1994.

[SP95] W. Richard Stevens and Jan-Simon Pendry. Portals in 4.4BSD. In *Proceedings of the 1995 Usenix Technical Conference*, pages 1–10, January 1995.

[SRY93] B. C. Smith, L. A. Rowe, and S. Yen. Tcl Distributed Programming. In *Proceedings of the Tcl/Tk Workshop*, June 1993.

[vRHB94] R. van Renesse, T. M. Hickey, and K. P. Birman. Design and Performance of Horus: A Lightweight Group Communications System. Technical Report TR 94-1442, Cornell, August 1994.

[Whi94] James E. White. Telescript technology: The foundation for the electronic marketplace. General Magic White Paper, 1994.