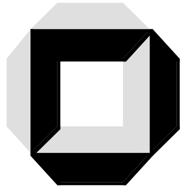
CORE
Metadata, citation and similar papers at core.ac.uk
Provided by KITopen



Universität Karlsruhe
Fakultät für Informatik

76128 Karlsruhe

Delegating Remote Operation Execution in a Mobile Computing Environment[1]

Februar 1996

Dietmar A. Kottmann
Ralph Wittmann
Markus Posur

Universität Karlsruhe
Institut für Telematik

Interner Bericht 9/96

[1]This work was partly supported by the German Research Council (Deutsche Forschungsgemeinschaft DFG) within the interdisciplinary research group SFB 346 comprising mechanical engineering and computer science projects under grant SFB 346–A6 and grant SFB 346–D4.

Abstract

Remote operation execution is nowadays the most popular paradigm used to build distributed systems and applications. This success originates in the simplicity exhibited by programming along the client–server paradigm. Unfortunately, connectivity and bandwidth restrictions defy the unchanged porting of this well known mechanisms to the mobile computing field.

In this paper we present an approach that allows to develop applications which are tailored for the specific requirements of mobile computing, while retaining the simple and well understood remote execution paradigm. The approach provides the additional benefit that established services could easily be used from mobile platforms. The cornerstone of our approach is integrated linguistic support for dynamically delegating the execution and control of remote procedure calls (RPC) to a delegate located on the fixed part of the network. Besides presenting the language constructs, we discuss the extensions to the RPC–based development process and the necessary run–time support.

1 Introduction

Performing remote operations is nowadays the prevailing paradigm for constructing distributed systems and applications [Tan95]. It is used in various flavors ranging from remote procedure call (RPC), as found in OSF DCE [BGH+93] or in ONC [Blo92] to remote method executing in object–based systems, like OMG CORBA [Omg93]. The attractiveness of the paradigm originates in its conceptual simplicity which is summarized under the term client–server–style programming[2]. All modern integrated environments for distributed systems, like ANSAware [Ans92], OSF DCE [BGH+93], OMG OMA [Omg92] or ISO ODP [ISO92], base on this paradigm. : It would be an attractive perspective to simply put applications which base on remote operations onto mobile platforms to get real mobile applications. Unfortunately, mobility has impacts on distributed systems that defy this direct approach, c.f. [FoZ94, BAI93]. In an nutshell, mobile applications have to face temporary disconnections, a high degree of bandwidth variability with long periods of low bandwidth, while operating on resource–poor platforms. None of these constraints could be expected to vanish through foreseeable technological progress [Sat93].

Using remote operations in this context has two major defects. Firstly, remote operations are synchronous in nature, leaving the client blocked until the operation has completed. A behavior clearly undesirable in environments that are prone to disconnections. Secondly, clients traditionally have complete control over the distributed computation. This leads to executions where clients obtain information from one operation only to use (parts of) the information as input for the next invocation. In this client–centered control philosophy the client is the link that glues the pieces of an application together. Thus, the entity that due to limited bandwidth and occurring disconnections is physically the weakest part plays logically the most important role for the application. On the other hand, it is important to enable mobile clients to make use of the existing infrastructure. Consequently, native access protocols for servers have to be retained.

In this paper we present an approach that accounts for the identified prerequisites. Mobile clients are given the possibility to specify a control policy and a corresponding information flow for multiple remote operations. The execution is asynchronously performed by a *delegate* on the fixed network. The delegate is able to return results to the client and to invoke callback–operations on the client actively, in case specified activation conditions are met. The delegate is represented on the client side by a *trustee* which performs a seamless linguistic integration. Our approach integrates the development process of delegates and trustees with the RPC–development cycle. However, our approach is general enough that it could be advantageous for any distributed computing environment which is based on remote operation execution.

The presentation is organized as follows: After giving a systematic overview over the problems of remote operations in a mobile computing context in section 2, we present our

[2]Note that client–server–style programming only refers to the idea of structuring a distributed system in entities offering services and entities consuming services. An entity is allowed to produce a service under usage of other (remote services). Dedicated client– or server–machines or –programs aren't necessary prerequisites.

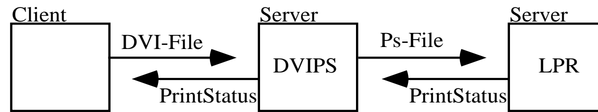


Figure 1: The client/server model

concept of trustees and delegates in section 3. This approach enables different programming styles well suited to mobile computing. These are discussed in section 4. The necessary system support for the development process and for run–time protocols is subject to section 5. In section 6 we report about initial experience and give some performance figures. Finally, section 7 discusses related approaches and section 8 concludes with a summary and a brief outlook.

2 The Case of Remote Operations in Mobile Computing

Remote operations are built around the client/server model (see figure 1): A server typically offers some operations to a community of clients. Clients then select a server that offers some desired functionality[3]. A client can use an operation by sending an invoke message to a suitable server. The message typically contains input parameters (e.g. data to be printed). After receipt of the invocation request the server performs the requested service and finally sends a result back to the client. Normally, the result contains some parameters (e.g. the print status).

As shown in figure 1, a server can itself act as a client with respect to another server. Such servers are called *complex servers*. Whether a server is complex or not is generally invisible to the client. The client isn't able to tell the complex server of figure 1 from an implementation where one server provides the integrated functionality to convert data from **dvi**– to **ps**–format and to put the resulted file to the printer.

As distributed environments grow, so do the number of operations which are offered to clients[4]. With this increase it becomes impossible to provide all sensible combinations of servers as hard–coded complex servers. Thus, clients dynamically combine servers when no single server offers a desired functionality. Consider the case depicted in figure 2. A client needs the functionality of the complex server of figure 1 but only finds a server that is able to perform the conversion from **dvi**– to **ps**– format and another one that prints ps–files. When the client wants to print a dvi–file, it combines the two servers dynamically. How the combination is performed is subject to the control policy coded inside the client program.

[3]The server selection — or binding — procedure could itself be performed via invoking a binding procedure on a dedicated server. Using this bootstrap mechanism, only a very small set of servers has initially to be known to the run–time system of a client.

[4]Note that operations are not restricted to the offers of conventional servers but are the general building blocks for distributed applications. Such application specific remote operations outnumber the generic services by far.

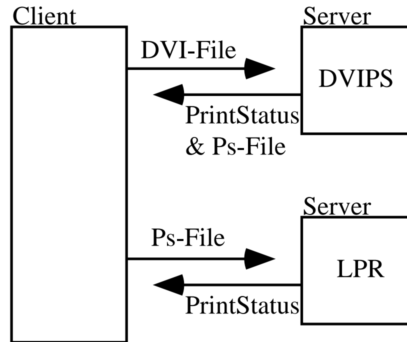


Figure 2: Client–centered construction of combined services

In addition, the client governs the information flow between different servers in mapping (parts of) the obtained result into new invocations.

The assumptions about remote operation execution, which underlay the above discussion, could be summarized as follows:

Fast networks and servers: Invoking remote operations blocks the client. Hence, both the network and the server have to perform their operation reasonably fast.

Reliable communication: When either invocation or result are lost, the client is doomed to wait forever or has to live with the danger of orphaned computations. Thus, the underlying network is assumed to be reliable, unless special application level functions to recover from broken communication channels are developed.

Several generic mechanisms to deal with failures have been invented. The most popular ones are Exactly–Once–RPCs that rely on transactions and At–Most–Once–RPCs that cope with several transient failures of the underlying transport system [Nel81]. Fault tolerance for idempotent services is achieved with At–Least–Once–RPCs [Nel81]. Additionally, several replication techniques for RPC systems have been explored (c.f. [YJT88, Coo84]). All these mechanisms are expensive. Hence, most RPC systems only offer At–Most–Once–RPCs as a generic mechanism, because transactions are too expensive for general use. Note that this mechanism does not defy orphaned computations in general, so that it is no solution for networks that are prone to disconnections. For the special but rare case of idempotent services, At–Least–Once–RPCs are also provided by many RPC systems.

Continuous network availability: Combining operations dynamically — for client–centered construction of combined services — relies on continuous communication capabilities.

Homogeneous network availability: Allowing clients to combine operations dynamically is only adequate when the chance of a client to reach a particularly server are about the same as the chances for a complex server to do the same job.

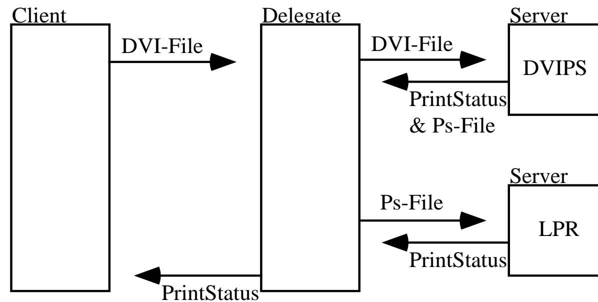


Figure 3: Delegating remote operation execution

All of these assumptions become invalid when one considers a mobile computing scenario with mobile clients. Here, communication between the client and server is generally slower as communication on a fixed network, more error prone, subject to disconnections, and inhomogeneous, as the chances to reach a server from a fixed node are better than the chances to reach the server from a mobile node. Note that this holds regardless of whether the server is placed on the fixed network or is a mobile node itself. The latter is true, as communication between two mobile nodes often has to pass two error prone low bandwidth links between the fixed network and the mobile nodes.

A countermeasure to these problems would be to increase the number of precompiled complex servers. Following this track means that starting a client implicitly starts and registers a huge number of complex servers on the fixed network, one for each conceivable combination of operations the client might invoke — regardless of the invocation–frequency. This explosion of the number of complex servers is clearly unbearable as in the worst case their number grows over–exponentially (in the order of n! — ignoring the additional possibility to define multiple parameter–mapping combinations) in the number of application servers (n).

We propose as an alternative the paradigm of delegating remote operation execution. Instead of precompiling a server for each conceivable combination the fixed network provides *delegates* which have the capability to execute combined operations dynamically according to a specification provided by the client. Instead of stashing the fixed network with a huge number of precompiled complex servers, the client carries the specification of all complex services it eventually needs. On demand the specification is uploaded to a delegate. Note that this approach is especially suited for the case of mobile clients that visit several fixed networks, as each client can dynamically instantiate the services it needs.

Now executions as shown in figure 3 result. The error prone low–bandwidth link from the mobile node to the fixed network is only stressed once for the initial invocation and once for the result message, while the operation invocations are performed completely inside the homogeneous fixed network. Note that there is no need to transfer the potentially big intermediate results or parameters over the low–bandwidth link. The resulting behavior is tailored to the requirements of mobile clients.

```
BEGIN
  DviResult := DviPs(DviFile, PsFile);
  CASE DviResult.errno OF
    0:
      Result.errno := Lpr(PsFile);
      RETURN(Result);
    | -1:
      Result.errno := -2;
      Result.msg := "Fatal DVIPS Error";
      RETURN(Result);
END;
```

Figure 4: A simple delegate

3 Delegating Remote Operation Execution

This section discusses how linguistic support for delegating remote operations can be integrated into a distributed computing environment. Although our approach is general enough to be applicable for each remote execution mechanism, an RPC–based solution is used for the presentation. First, we discuss the method to specify control policies and information flows for *delegates*. Afterwards, *trustees* are introduced as a mechanism to invoke delegates asynchronously. An extended example of how both are used is postponed to section 5.5.

3.1 Delegates: Controlling Operations Remotely

Delegates have to exert control over multiple remote operations on behalf of the client. Additionally, they have to channel the information flow between multiple operations, in mapping output parameters to input parameters of subsequent calls. Both is most easily achieved by using an operational description.

A simple example of how delegates are defined is given in figure 4 which defines the control flow of the printer example. For brevity type definitions are omitted. A more elaborate example could be found in section 5.5. An example for type definitions is given in section 5.3.

The language DIL (**D**elegate **I**nstruction **L**anguage) to specify delegates comprises an `IF`, a `WHILE`, a `CASE`, and a `FOR` construct as simple control structures and a dedicated type system that allows to catch the whole semantics of the basic RPC system. It allows to specify arbitrary control policies and arbitrary information flows between subsequent operation invocations.

```
Trustee_pt TrusteeDispatch_PrintDvi(File* DviFile);
Result_t   TrusteeFetch_PrintDvi(Trustee_pt Trustee, int Indication);
BOOLEAN    TrusteeIsReady(Trustee_pt Trustee, Time t);
BOOLEAN    TrusteeDiscard(Trustee_pt Trustee);
```

Figure 5: Basic features of a Trustee

3.2 Trustees: Asynchronous Invocation Processing

Blocking the client program until a complex service completes disqualifies the remote operation execution paradigm for a broad range of applications. The natural countermeasure is to provide a mechanism for performing operation invocations asynchronously with the possibility to defer synchronization with the operation result to some later time.

The need for such a mechanism grows with the execution time of a remote operation. Combining multiple operations in one delegate increases the mean time to completion and makes a mechanism for asynchronous operation execution even more necessary.

Our approach to asynchronous operations are *trustees*. Trustees borrow their basic functionality from the *future* concept of Cronus [WFN90] but employ several important extensions which are especially suited for the mobile computing context.

3.2.1 Trustees: The Basics.

Consider a remote procedure that prints a dvi–file and returns a print status:

```
Result PrintDvi(File* DviFile);
```

The basic operations of the corresponding trustee are shown in figure 5[5]. The remote operation is asynchronously invoked by calling the RPC–specific **TrusteeDispatch** operation. This procedure immediately tries to forward the invocation over the network. Regardless of the success of putting the message on the network the operation immediately returns with a reference to a newly generated trustee object. This object serves as a placeholder for the call. As placeholders are dynamically allocated multiple calls to one remote operation could be outstanding at a time.

The reference to a trustee could be used for multiple purposes. Firstly, if the application decides that it won't need the result of the operation later on, it can notify the system that the placeholder need not longer be maintained by calling the **TrusteeDiscard** operation. Secondly, the application can test whether a valid result has already been received via the **TrusteeIsReady** operation. An optional parameter allows to specify a time span for which the operation is allowed to block. The operation returns when a valid result has been received or when the specified time has passed. The default time span makes the

[5] In the remainder of this paper examples of stub procedures and interfaces will be presented using the C programming language, although the trustee abstraction is independent of C.

```
DelegateInterface PrintDvi
     IMPORT ...
     USES ...
     TYPE ...
     IN
       File*     DviFile;
     LOCAL_IN ...
     OUT
       Result_t  Result;
END.
```

Figure 6: A delegate definition frame

call return immediately. Finally, the RPC–specific **TrusteeFetch** allows the application to block until a result has been received. This way, synchronization with outstanding calls is achieved.

Note that the operations **TrusteeDiscard** and **TrusteeIsReady** are generic. Only the other two operations **TrusteeDispatch_<Service>** and **TrusteeFetch_<Service>** are specific for each involved RPC procedure.

3.2.2 Combining Trustees with Delegates

Basically, trustees are part of the run–time system of the client. They allow applications to defer result handling for RPC calls to the time when the RPC has completed or when its results are needed. Thus, the control still conforms to the conventional client/server paradigm (c.f. figures 1 and 2).

Combining trustees with delegates allows for deferred synchronization with the results of delegated complex operations. This integrated model is based on defining delegates in a delegate definition frame. The frame for the delegate code of figure 4 is depicted in figure 6. The corresponding trustee has the same interface as the one for the simple RPC–call of section 3.2.1 (c.f. figure 5).

Note that a delegate that simply performs one RPC call can have the same trustee interface as one that combines basic operations dynamically. Thus, it is transparent to an application which uses a trustee for deferred synchronization whether it uses a complex delegate or a simple RPC operation.

3.2.3 Trustees — Advanced Features

The so far discussed operations of the trustee interface build an orthogonal syntactic framework that enables asynchronous remote operation invocations. Still it is not sufficient to model typical problems of the mobile computing context in a straightforward manner.

In many cases mobile nodes serve as presentation front–ends. Their integration in the

distributed environment allows them to access shared databases or general server facilities. It is often the case that a user wants to delegate some task to the system and that he trusts the system that the task will be handled correctly. The system should only disturb him when an exceptional condition arises. Consider once again the printer example. After invoking the print operation the system should do its job quietly, unless the print fails. In that case the user should be notified by a message box that pops up on his display and gives him the means to inquire what exceptional condition has occurred.

Although the desired behavior can be realized using only basic trustee functions, this defies a clear and modular design of client applications. All parts of the application program would have to be aware of the outstanding print operation to poll the trustee via the **TrusteeIsReady** operation again and again, only to find that in most cases the print has been completed successfully. So all subroutines that allocate trustees would impose the constraint of polling and deallocation onto all other routines. To enable a clearly separated and modular design it is necessary to equip the trustee with a callback interface which can be invoked by the delegate under certain conditions.

A callback behavior is achieved by the delegate depicted in figure 7. The corresponding trustee (figure 8) enables to specify a reference to a callback function. The extended **TrusteeDispatch** operation comprises the input parameters of the delegate together with references to the local callback functions. The **TrusteeFetch** operation corresponds once again to the output parameters of the delegate.

Note that the specified callback function does not contain a description of the exception directly (e.g. as a string). Instead, a reference to the trustee has been included. Directly coding the exception condition into a callback parameter would also have been possible. We used the depicted alternative to show how the callback mechanism and the return value are used in an integrated manner. Here the callback pops up a message box, which enables the user to inquire the exceptional condition. In case the user wants to find out what exception did occur the callback invokes the **TrusteeFetch** operation on the returned trustee reference. Note also that the callback references a local variable, namely the reference to the window system that is only valid on the client node. As there is no need to send this reference over the network it is retained inside the trustee until needed. This is the difference between the `IN` section and the `LOCAL_IN` section of the delegate definition framework. Both define input parameter for the **TrusteeDispatch** operation but only the ones defined in the `IN` section are transferred over the net. Hence, the delegate code must not use the variables defined in the `LOCAL_IN` section, besides as parameters for callback functions (c.f. figure 8).

4 Programming with Trustees and Delegates

In this section three basic ways to program trustees and delegates are introduced.

```
DelegateInterface dvilpr
     IMPORT ...
     USES ...
     IN
       File*     in_f;
     LOCAL_IN
       Obj_t     MsgBox;
     OUT
       Result_t  Result;
     CALLBACK
       void      OpenMsgBoxForTrustee(
                   Obj_t, Trustee_pt
                 );
END.

DelegateImplementation dvilpr
     BEGIN
       DviResult := DviPs(in_f, ps_f);
       CASE DviResult.errno OF
         0:
           Result.errno := Lpr(ps_f);
           RETURN(Result);
         | -1:
           Result.errno := -1;
           Result.msg := "Fatal DVIPS Error";
           OpenMsgBoxForTrustee(MsgBox, self);
           RETURN(Result);
     END.
```

Figure 7: Delegate with a callback function

4.1 Conventional asynchronous programming

The most simple way of using trustees and delegates is to take them as an alternative to existing asynchronous RPC systems. How this feature can be employed is well known (c.f. [WFN90, AnT92]). The additional feature of our approach is the possibility to incorporate complex services that are constructed from existing off–the–shelf servers. We omit the further discussion, as those complex servers are used in exactly the same way as monolithic ones.

```
Trustee_pt   TrusteeDispatch_dvilpr(
                  DviFile*  fd,
                  Obj_t     MsgBox,
                  void      (*OpenMsgBox)(Obj_t,Trustee_pt)
             );
Result_t     TrusteeFetch(Trustee_pt Trustee,int Indication);
```

Figure 8: Trustee for a delegate with a callback function

```
...
Trustee_pt MyTrustee = TrusteeDispatch_dvilpr(MyFile, MyMsgBox, MyCallback);
TrusteeDiscard(MyTrustee);
...
void MyCallback(Obj_t MsgBox, Trustee_pt Trustee) {
...
}
```

Figure 9: Fire–and–Forget programming style

4.2 Fire–and–Forget RPC

Trustees and Delegates can be used for the straightforward solution of what we call fire–and–forget problems. In those problems, the client invokes a complex service and relies on the system to bring the invocation to a good end and to do any necessary housekeeping automatically. The user should only be involved when some exceptional condition occurs, like in the motivating example for the callback facility, presented in section 3.2.3.

Take figure 9 as an example. Directly after invoking the delegate of figure 7 asynchronously, the program calls **TrusteeDiscard** on the returned trustee pointer. Hence, the main program does not want to use the trustee anymore. However, the run time system only marks the trustee as discarded, because the delegate might invoke the callback procedure. Now two alternatives are possible. The first is that the delegate completes without invoking the callback. When the run time system on the client gets this reply, it automatically deallocates the trustee and performs all additional housekeeping. Now assume the delegate invoked the callback. When the run time system on the client gets this reply, it invokes the callback function, which could synchronize onto the returned results, through the pointer to the trustee which is used as a parameter in the delegate code in figure 7. Invoking the trustee from the delegate is legal because the trustee is still allocated and only marked as being discarded. When the callback function has been completed, the run time system deallocates the trustee and performs all housekeeping, as the trustee had been marked as discarded.

This way, the common case that one wants to invoke a remote service and only wants

```
...
Trustee_pt MyTrustee = TrusteeDispatch_dvilpr(MyFile, MyMsgBox, MyCallback);
...
if(TrusteeIsReady(MyTrustee))
     result = TrusteeFetch(MyTrustee, ind);
TrusteeDiscard(MyTrustee);
...
void MyCallback(Obj_t MsgBox, Trustee_pt Trustee) {
...
}
```

Figure 10: Integrated Callback Processing

to be interrupted in case something goes wrong can easily be achieved with high level linguistic support. The code that invokes the trustee never needs to poll or discard it in the future, as the immediate discard is postponed by the run–time system to enable later callbacks. In case everything ends well, the postponed discard becomes effective. When an exceptional condition occurs, the callback is invoked to report the exception. After the callback handling completed, the postponed discard becomes operational. This way, housekeeping is performed at the right moment without the need to use low level code.

We believe that this style of programming is common for the mobile computing field, as notebooks often serve as presentation front ends. The trustee and delegate concept is the first high level linguistic support to this style of programming without the need to use a low level interface, like a thread library.

4.3 Integrated Callback Processing

The fire–and–forget programming style can be supplemented by elements of the asynchronous programming paradigm. Here, the main program can synchronize on the results of the asynchronous invocation. Besides this exceptional conditions can be reported actively via callbacks. A simple example for this style of programming is given in figure 10.

The trustee remains once again allocated until the `TrusteeDiscard` operation is invoked. Hence, both the callback and the synchronization in the main code operate on a legal trustee reference. This style of programming can for example be employed when the main program is an interactive application itself. Here the callback can be used to report to the user that some operation has been completed. The user can decide whether he wants to follow the notification immediately to inquire the results or to complete his current task before looking at the results.

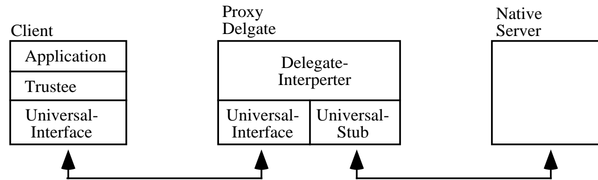


Figure 11: Overall system architecture

5 System Support

In this section the system architecture is described. Thereafter a development cycle is pointed out which is similar to the development process in other RPC systems. A brief description of implementation aspects of delegate proxies and trustees follows. The section closes with an example of DIL.

5.1 System Architecture

The overall system architecture is depicted in figure 11. The delegate itself is an interpreter which executes a script downloaded with an invocation message. How these scripts can be derived from a DIL–specification is explained in the next subsection. The client side of the architecture comprises a run–time incarnation of trustees which serve as equivalents to the well known RPC–stub concept. The trustee communicates via an universal stub with the proxy delegate. Invocations, results, callback invocations, and interpretable delegate code are transferred over that stub. This is described later in this section. The interpreter uses a universal interface to mimic general remote procedure calls. This way native servers can be accessed unmodified. Note that this allows to introduce trustees and delegates into an environment comprising many existing services.

The system consists of three parts:

- DC, a compiler generating delegate scripts from DIL–specifications,
- a proxy delegate,
- a trustee run–time library.

5.2 The Development Cycle

RPC–based systems have become popular not only because of the simplicity of the underlying client/server paradigm but also because of an development cycle which allows the integrated development of remote applications. To keep this benefit we have to define an integrated development process for systems comprising trustees and delegates. The resulting cycle is given in figure 12.

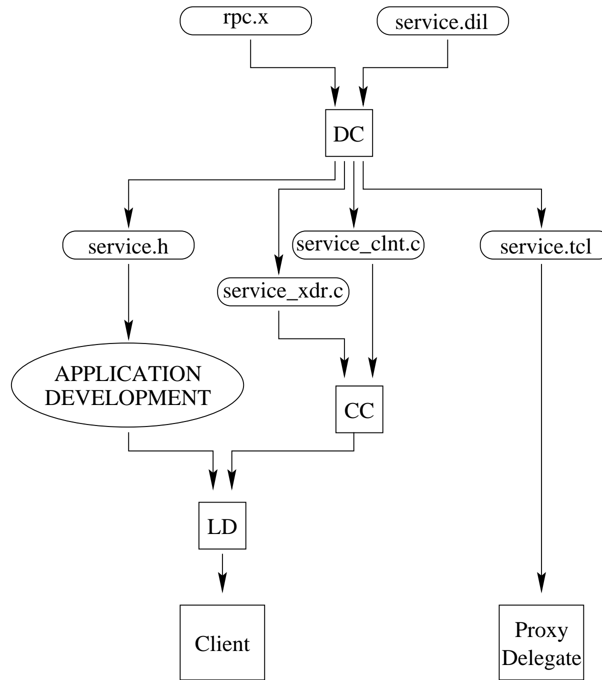


Figure 12: The integrated development cycle

The development process rests on the type declarations which are employed by the client and the interface definition files, normally used by the basic RPC system for generating RPC stubs. Together with the delegate–definition written in **DIL** (Delegate Instruction Language) they form the input of the *Delegate Compiler* ***DC***. DC produces an interpretable code which can dynamically be installed on the proxy delegate at the fixed network. The other files produced by DC are used in developing the client application. One of them is a header file which defines the trustee interfaces for all delegates. The other two implementation files code type specific trustee–stub routines and stubs implementing the universal interface.

5.3 The Proxy Delegate

Implementing the RPC–based realization of the concepts has to be tailored tightly to the RPC system which is employed by the servers. We chose ONC RPC as our validation vehicle because it is easy to access basic run–time routines of the ONC package via well documented interfaces [Blo92]. This is necessary to allow the proxy delegate to mimic a real client stub via its universal stub.

As the basic code interpreter for the proxy we use Tcl [Ous94]. We extended the interpreter with code to access the universal interface, with code that allows for the management of explicit type information as needed by the RPC system, and with routines that allow the easy mimicking of RPC stubs. An example for a Tcl script produced by our compiler DC can be found in section 5.5.

```
ClntCall  $ProcName $ServerName $Input $Input_t \
        Output Output_t
```

Figure 13: RPC with Tcl

With our extensions a call to a remote procedure in Tcl looks like the fragment depicted in figure 13. All input and output parameters are coded into one complex parameter as usual for ONC RPC [Blo92]. Each parameter is followed by its type descriptor to allow the coding into or from the external data representation XDR [Sun87] used by the RPC system.

The Function **ClntCall()** uses the explicitly given type descriptors for the parameters to construct complex XDR filter routines for parameter–marshalling at run–time from basic XDR filters that marshal simple types like integers. This way, arbitrary clients are mimicked at run–time, so that native RPC servers can be accessed. Note that we cannot simply use compiled complex XDR filters as common RPC stubs do, because at compile time the set of servers that will be accessed via the proxy delegate is unknown. Hence, using complex XDR filters would need system support to install code dynamically.

The basic XDR filters are dynamically combined as follows. A *type descriptor* for use in the proxy delegate represents a type definition in a nested list structure. A symbol that does not code a basic type like an integer for which a XDR filter exists marks another type descriptor. This way recursive definitions are structured. An example is given in figure 14. It depicts two type definitions in XDR and their corresponding type descriptors generated by DC. As the symbol *exp_t* in *u_t's* descriptor does not stand for a basic type it refers to a complex type descriptor. Note that DIL uses XDR style type definitions.

With these Tcl–extensions a delegate invocation proceeds as follows. After recepting an invocation message a delegate server creates a Tcl interpreter which is initialized with the delgate script. The script contains type descriptors which are used to unmarshall input paramters. Thereafter the control is given to the interpreter to execute the script. Every ClntCall–command encountered by the interpreter causes an RPC. Once again the type descriptors are used for marshalling and unmarshalling of input parameters and results. Finally, the results and are marshalled and sent back to the client trustee.

5.4 Run–Time Support

A trustee object consists of two parts, stubs and run–time routines. The stubs are created by DC according to the interface definition written in DIL. To simplify stubs and compiler most of the trustee tasks are handled by run–time routines. Stubs provide a convergence layer to the library functions. The specific trustee functions are mapped to a small set of run–time routines handling delegate RPC calls and results and managing callbacks. Furthermore the run–time system takes care of allocating and releasing of trustee resources.

To allow for asynchronous invocation of trustee–operations and deferred result handling

```
union u_t switch (int errno) {
     0 : int   status;
     1 : exp_t member;
};

struct exp_t {
     int    i;
     string s<20>;
     float  f;
};

u_t     {union {int {0 int} {1 exp_t}}}
exp_t   {struct {int {string 20} float}}
```

Figure 14: Type descriptions in XDR/DIL and Tcl

the trustee–code is multi–threaded. The run–time routines provide for synchronization of callbacks and trustee operations. Being multi-threaded trustees need more support by a operating system than the proxy delegate. Since there is no standard interface for threads except POSIX which is not available for all platforms an independent interface is used. Accordingly, any operating systems supporting some kind of threads can be used by providing that interface. Using the widely held SUN–RPC and Tcl which are available on various platforms our the delegate implementation is far–reaching portable. Currently we have implemented our system on plattforms running OSF/1, SunOS 4.1, Solaris 2.3, and Linux 1.1.13.

5.5 Example

In this section an elaborated example is given which demonstrates how to use DIL. Reconsider the dvi–to–ps example depicted in figure 3. A simple DIL–description implementing that task as a fire–and–forget RPC is shown in figure 15 and 16. *DviPs* and *Lpr* are the native services which are accessed via the delegate. They are defined in the RPCL specification *dvi.x* imported by the definition frame. RPCL is a definition language used by ONC-RPC to define service interfaces and types (c.f. [Blo92]). A service definition written in RPCL can be used with DIL without changes. This service gets *in_f* as an input parameter, its type is defined in TYPE section. The service *DviPs* is called to produce a postscript representation of the input. If that service succeeds the *LPR* service is called to print the file. If no error occurs the service is executed silently. Otherwise a callback procedure 'Alert' is invoked which may notify the error to the user. It takes a string as an input parameter. This string is transferred to the calling trustee which invokes the callback. Note that the native server hosts *nemesis* and *kastor* are hard coded in this example for

```
Delegate Interface PrintDvi
IMPORT
     "dvi.x";
USES
     DviPs;
     Lpr;
TYPE
     string str_t<>;
IN
     str_t in_f<>;
LOCAL_IN
OUT
CALLBACK
     void Alert (str_t);
END.
```

Figure 15: DVI–PS filter interface

```
Delegate Implementation PrintDvi
DECLARE
     str_t   ps<>;
     int     result;
BEGIN
     ps := DviPs("nemesis", in_f);
     IF ps = "" THEN
        Alert("DVI-PS ERROR");
        RETURN(NULL);
     END;
     result := Lpr("kastor", ps);
     IF result = -1 THEN
        Alert("LPR-ERROR");
     END;
     RETURN(Result);
END.
```

Figure 16: DVI–PS filter in DIL

brevity, but more sophisticated name schemes can be applied.

The corresponding Tcl-script dedicated to a proxy delegate depicts figure 17. The first four lines define the type descriptors for type *str*, the input and output parameters — there

```
set str_t          {string max}
set arg_in         {{str _6} }
set arg_out        {}
set arg_cb_Alert   {str_t}
set arg_result     {union {int {0 arg_out} {1 arg_cb_Alert}}}

proc _main {} {
     upvar 1 _6 _19
     set _11 [ClntCall ProcNr_1 "nemesis" tcp $_19 str_t _20 str_t]
     if {$_11 == ""} {
       Alert 1 "DVI-PS ERROR"
     }
     else {
       set _12 [ClntCall ProcNr_2 "kastor" tcp $_11 str_t _21 int]
       if {$_12 == -1]} {
          Alert 1 "LPR ERROR"
       }
     }
}
```

Figure 17: Delegate–script for DVI-PS filter

is no output parameter in this example —, and the signature of the callback procedure. The fifth line describes the format of the result message, a simple integer serves as a type discriminator to let the trustee know whether he has to invoke a callback just deliver the result parameter. The remainder of the script is a straight forward transcription of the DIL program given above.

6 Evaluation

In this section, we assess the performance of our approach. Using an interpreter instead of precompiled modules introduces execution overhead. Firstly, the invocation message contains not only the service arguments but the delegate script. Moreover, every instruction has to be parsed and interpreted each time it its encountered. As Tcl provides only the string type every data object has to be converted into a string representation. This is done at the time a call arrives at the proxy delegate. When the delegate calls a native server the inverted conversion takes place. A delegate can only become faster when the complex service it defines saves RPCs over the slow wireless link and allows to keep intermediate parameters on the fixed net, as in the dvips example. As the superiority of the delegate approach for very slow low bandwidth links is obvious, we were interested on its performance in the area of considerable fast radio LANs. Note that this is the worst case scenario for the delegate

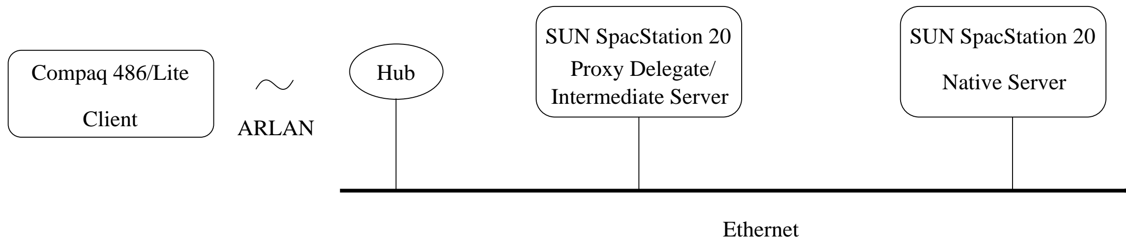


Figure 18: Second Scenario

approach.

To assess performance the resulting degradation we used two different scenarios. In the first scenario a native ONC–client with a corresponding native ONC–server were compared to a call from the Tcl interpreter through our universal stub `ClntCall` to the same native ONC–server. We measured how long a call parameterized with an array of integers lasts. The server only printed the number of bytes received. As a mobile station we used a Compaq 486DX2/Lite Notebook equipped with 4 MB main memory, MS–Windows 3.11, and a ARLAN PCMCIA adapter that was configured to provide a 1 mbps radio link. The proxy delegate was located on a SUN SparcStation 20 with Solaris 2.3, the native server did run on another SUN SparcStation 20. The testbed structure is shown in figure 18. The results are given in figure 19. Note that this is basically an unfair comparison. The native call only has to pass the stub on the client side and demarshalling procedure on the server side. The delegate approach first has to pass the stub on the client side, then the demarshalling on the proxy delegate, the interpretation inside the delegate, the universal stub of the delegate and finally the demarshalling at the native server. Thus, the delegate approach must perform worse. It can only develop its strength when a real complex service with intermediate parameters is used.

To compare our approach to an approach using precompiled modules, an intermediate server was added to the scenario. That server simply passed the calls form the client to the other server. This result is also included in figure 19. The comparison is once again unfair, as a simple pipeline module naturally has to perform better than an interpreter that decodes and encodes the parameters. All in all, this comparison revealed that the interpreter approach has about half the performance of the native pipelining approach.

To infer the reason for this degradation, we made another test that used more modern equipment. So we compared the interpreter approach to the native approach on two modern DEC AXP 3000/800 workstations on a lightly loaded Ethernet. The obtained results are given in figure 20. The figure shows that the overhead is in the order of 50%. The overhead is bigger for small transfers, as the interpreter imposes a basic constant overhead. Unfortunally the lab with the AXPs is currently not equipped with a wireless link to enable a direct comparison to figure 19.

A more elaborate interpretation of figures 19 and 20 now unreveals the reasons for the performance degradation in the first scenario. Firstly note, that a native Null–RPC between two AXPs did last about 2ms, while the native Null–RPC between the mobile

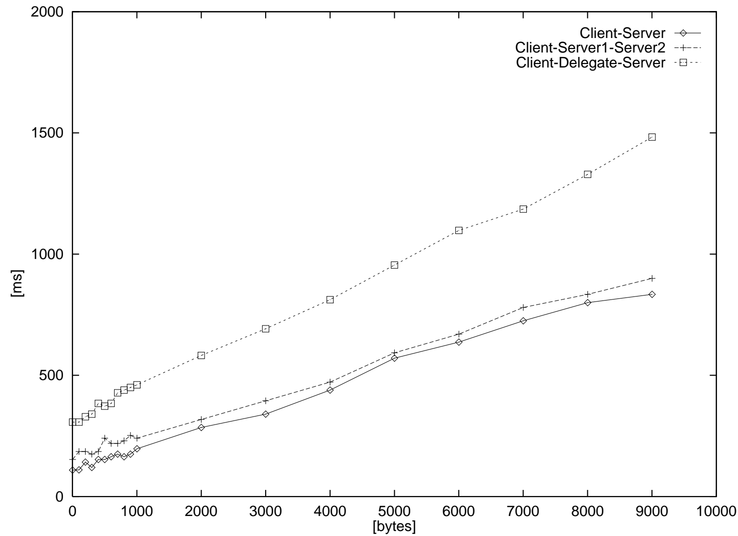


Figure 19: Comparison between delegates and compiled servers

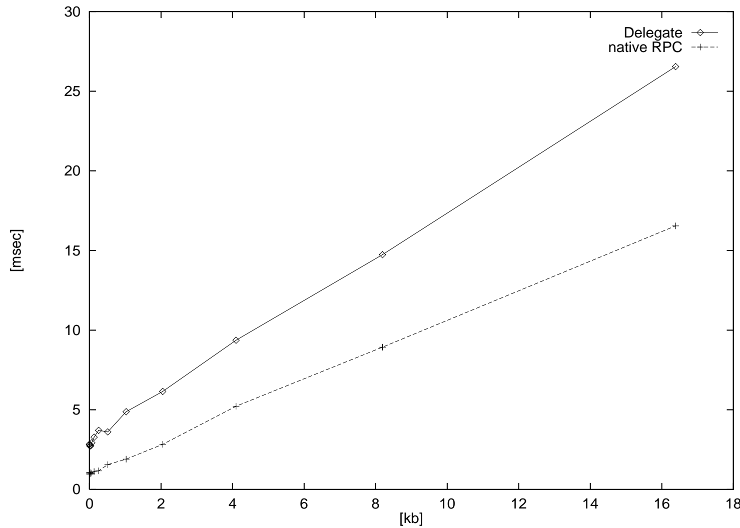


Figure 20: Comparison between delegates and native RPC

client and the fixed server (i.e. without delegate) did last about 110ms. The Null–RPC with the intermediate Pipeline did cost 150ms. Hence, marshalling and unmarshalling can last at most 20ms on the Suns. So parts of the difference between the 2ms and the 110ms can be blamed to the slower workstation. However, a difference of about 70ms between the wireless Null–RPC and the Ethernet Null–RPC remains. This means that delegation is in principle a good idea, as the difference between the wireless and the fixed network performance is worth the expense to run an interpreter on a fast workstation and delegation reduces the number of mobile RPCs for the price of interpretation.

Secondly we compare the two figures for a payload of 9KB. The Client–Delegate–

Server performance is 590ms. This comprises unmarshalling and marshalling again at the delegate. The basic overhead for installing the delegate is equal to the duration of the Null–RPC of 150ms in the client–delegate–server scenario. The difference, i.e. 440ms is due to the processing overhead for the marshalling and unmarshalling in the client, delegate, and server and in addition for the interpretation overhead. When we compare this to the complete time for an Delegate–RPC between two AXPs, which is 7ms, it becomes clear that most of the overhead results from the slow workstations used in figure 19.

All in all, the so far obtained results give us a strong feeling that the performance degradation for the delegate approach will vanish with modern hardware. Unfortunately we are currently unable to produce the figures for the inclusion of AXPs as proxy servers. But we will give them in the final version of this paper, as our own mobile computing lag is currently build and will become operational at latest in November. Then we will not only run the same tests as in figure 19 to infer the real impact of the slower hardware but also measure break–even points for the size of intermediate parameters when a complex service realized in a delegate becomes faster as the use of independent RPCs over the wireless link.

7 Related Work

Most of the related work is found in the context of asynchronous RPC systems. An overview of asynchronous RPC systems is given in the survey paper [AnT92]. As mentioned above, our basic trustee functionality is borrowed from the futures approach of the Cronus System [WFN90]. Also, ASTRA [ATK91], a system employing RPC as a general structuring paradigm for interprocess communication, is close to the basic functionality. None of the known approaches integrates a mechanism which is similar to our callback integration. Although, callbacks are sometimes used to extend RPC facilities — OSF DCE [BGH+93] being one of the best known examples — those callbacks only provide a server with a procedure handle on the client side. The call to the procedure and the correct handling of the callback reference has to be programmed into the server code explicitly. Those callbacks are generally used to request additional information from the client along an upcall philosophy. Thus it is impossible to use off–the–shelf servers and equip them with callback procedures to signal certain exceptional conditions to the client. Hence, programming styles like the fire–and–forget approach are hard to realize. But those features make our approach attractive for use in the mobile context. Furthermore, the handling of both local and remote references in the trustee is a unique feature that facilitates the task to develop clients well tailored for the mobile computing context. Those features could be emulated in using conventional synchronous RPC and threads (e.g. in OSF DCE [BGH+93]) for parallelism in the client. Although the functionality could be achieved, the level of abstraction is lower by far. Especially synchronization between the threads and integration of callbacks has to be handcrafted for each invocation. Thus the trustee and delegate concept provides an abstraction to keep the client programmer from steadily reinventing the wheel.

Basically, the combination of trustees and delegates could be seen as form of agent–based–computing (ABC, c.f. [Woo94, WoJ95]). Indeed, the idea to ship scripts over a

network for dynamically installing programs on the fixed network is also found in *Telescript* [Whi94, Hou94] or the *Messenger* approach [Tsc94]. Although, we share the central idea to delegate a script to the fixed network and to defer synchronization on the results until they are needed, the known script approaches aren't well integrated in an existing service market and don't provide a seamless linguistic integration. Moreover, no integrated development cycle is given and scripts cannot become active with respect to the client as is possible through our callback interface. Finally, the focus to use an generic interpreter to dynamically build complex RPC–operations based on non–modified RPC–servers, is novel.

Modeling complex services has also been discussed in the distributed systems management community [Kel94]. So called complex services were employed to achieve fault tolerance through dynamic reconfiguration in case parts of the server population goes down during an execution. Contrary to our approach, operations aren't combined to serve the specific needs of clients. The definitions of complex services have to be given by an administrator who instead writes executable programs that define the mapping between simple and complex services. Once again, the seamless integration is missing.

Finally, our ideas are related to workflow management systems (c. f. [CaB91]). A delegate can be seen as an entity that implements a short time workflow that is executed inside the fixed network. However, the similarities are only superficial. Instead of specifying long–lasting workflows which comprise several elementary business activities, we dynamically define short combinations of existing operations. Hence, graphical specification facilities and dedicated run–time support comprising databases, application specific tools, and groupware systems which form the core of workflow systems are irrelevant to our problem of delegating program execution. On the other hand, the features which are necessary to enable the delegation of remote operations execution in mobile computing — namely seamless linguistic integration, dynamic installation of operation services, and deferred synchronization — are irrelevant to the problem of combining business activities to long–lasting workflows.

8 Conclusions and Outlook

In this paper we presented an approach to delegate the execution of dynamically composed remote operations from a mobile client to a generic proxy server on the fixed network. This allows for combing off–the–shelf servers as needed by clients, tailored to the specific requirements of mobile computing like disconnection prone links with low bandwidth. We discussed how two run–time entities, the trustee on the client–side and the delegate on the side of the fixed network, are used to provide a homogeneous and easy to integrate enhancement to existing distributed computing environments. Furthermore, we reported about our current initial prototype implementation, basing on ONC RPC, and our initial experience.

One open question is the problem of whether there exist implicit methods to control the lifetime of delegates. Although the presented explicit primitives offer straightforward solutions for most cases, some programmers may have difficulties to use them correctly.

Performance of the proxy delegate could be improved, if we only copied data needed by the delegate code in its Tcl representation. Doing this requires data flow analysis inside DC. We plan to exploit this possibility to test whether the improvements are worth the expense.

As delegate code is carried inside mobile clients and dynamically installed in the proxy delegates on demand, the concept can be exploited to carry client–specific complex services into visited host networks. Our current implementation does not comprise security measures to prevent clients form installing their delegates. Definitely one has to include means for authentication and accounting before qualifying the concept for everyday usage.

Finally, we plan to assess how trustees and delegates could be integrated with replication mechanisms that allow disconnected operations, c.f. [KiS91]. When a seamless integration becomes possible, some invocations on trustees needn't be forwarded to the delegates immediately. Instead of following the immediate transfer philosophy, the system could exploit bandwidth variations to choose transfer strategies and lifetime specifications for delegates according to the current situation on the network.

Acknowledgements

We are grateful to Jörn Hartroth for his detailed comments on an earlier version of this paper. We are also extremely indebted to the team of the mobile computing laboratory of Daimler Benz — especially to Norbert Diehl and Torsten Reigber — for allowing us to use their facilities as our test platform.

References

[Ans92] Architecture Projects Management Ltd. *ANSAware 4.0 Application Programmer's Manual*, March 1992

[AnT92] A. L. Anada, B. H. Tay: *A Survey of Asynchronous Remote Procedure Calls*, Operating Systems Review, Vol. 26, No. 2, April 1992, pp. 60-81

[ATK91] A. L. Anada, B. H. Tay, E. K. Koh: *ASTRA — An Asynchronous Remote Procedure Call Facility*, Proc. 11th International Conference on Distributed Computing Systems (ICDCS–11), Arlington, Texas, May 1991, pp. 172-180

[BAI93] B. R. Badrinath, A. Acharya, T. Imielinski: *Impact of Mobility on Distributed Computations*, Operating Systems Review, Vol. 27, No. 2, August 1993, pp. 15–20

[BGH+93] M. Bever, K. Geihs, L. Heuser, M. Mülhhäuser, A. Schill: *Distributed Systems, OSF DCE, and Beyond*, in: A. Schill (Ed.): *DCE - The OSF Distributed Computing Environment: Client/Server Model and Beyond*, Springer, Lecture Notes in Computer Science, No. 731, Berlin, 1993, pp. 1–20

[Blo92] J. Bloomer: *Power Programming with RPC*, O'Reilly & Associates, Sebastopol, California, 1992

[CaB91] J. C. McCarthy, W. M. Bluestein: *The Computing Strategy Report: Workflow's Progress* Forrester Research Inc., Cambridge, 1991

[Coo84] E. C. Cooper: *Replicated remote Procedure Calls*, Proc. 3rd Annual ACM Symposium on Principles of Distribued Computing, Vancouver, Canada, August 1984, pp. 220–232

[FoZ94] G. H. Forman, J. Zahorjan: *The Challenges of Mobile Computing*, IEEE Computer, Vol. 27, No. 4, April 1994, pp. 38–47

[Hou94] J. v. Houdt: *Personal Telecooperation Assistance*, In: K. Brunnstein, E. Raubold: *Proc. 13th World Computer Congress*, Elsevier Science B. V. (North-Holland), August/September 1994, pp. 11-17

[ISO92] ISO/IEC CD 10746–3: *Information Technology — Basic Reference Model of Open Distributed Processing*, December 1992

[Kel94] L. Keller; *Trading of Complex Services in Distributed Systems*, 5th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM'94), Toulouse, France, October 1994

[KiS91] J. J. Kistler, M. Satyanayanan: *Position Paper: Transparent Disconnected Operations for Fault–Tolerance*, Operating Systems Review, 25(1), January 1991, pp. 77–80

[MoS93] Markus U. Mock, Alexander B. Schill: *Design and Implementation of Distributed C++*, In: Peter P. Spies :*Proc. Euro–Arch'93*, Springer–Verlag, München, October 1993, pp. 469–482

[Nel81] B. J. Nelson: *Remote Procedure Call*, Ph.D. Dissertation, Carnegie–Mellon University, May 1981

[Omg92] The Object Management Group Inc.: *Object Management Architecture*, OMG TC Document 92.11.1, September 1992

[Omg93] The Object Management Group Inc.: *The Common Object Request Broker: Architecture and Specification*, OMG Document No. 93.12.43, Revision 1.2, 1993

[Ous94] J. K. Ousterhoust: *Tcl and the Tk Toolkit*, Addison–Wesley Professional Computing Series, Reading, MA, 1994

[Sat93] M. Satyanayanan: *Mobile Computing*, IEEE Computer, Vol 26, No. 9, September 1993, pp. 81–82

[Sun87] Sun Inc. *RFC 1014: XDR — External Data Representation Standard*, 1987

[Tan95] A. S. Tanenbaum: *Distributed Operating Systems*, Prentice Hall International Inc., Englewood Cliffs, New Jersey, 1995

[Tsc94] C. F. Tschudin: *An Introduction to the M0 Messenger Language*, Technical Report 86, Centre Universitaire d'Informatique, University of Geneva, May 1994

[WFN90] E. F. Walker, R. Floyd, P. Neves:*Asynchronous Remote Operations in Distributed Systems*, Proc. 10th International Conference on Distributed Computing Systems (ICDCS–10), Paris, France, May/June 1990, pp. 253-259

[Whi94] J. E. White: *Telescript Technology: The Foundation for the Electronic Marketplace*, General Magic White Paper, 1994

[WoJ95] M. J. Wooldridge, N. R. Jennings: *Agent Theories, Architectures, and Languages*, In: M. J. Wooldridge, N. R. Jennings: *Interacting Agents — Theories, Architectures, and Languages*, Springer Verlag, Berlin, Lecture Notes in Computer Science, No. 890, January 1995

[Woo94] A. Wood: *Agent-Based Interaction*, Internal Report, University of Birmingham, School of Computer Science, May 1994

[YJT88] K. S. Yap, P. Jalote, S. Tripathi: *Fault Toerant Remote Procedure Call*, Proc. 8th International Conference on Distributed Computing Systems, San Jose, California, USA, June 1988, pp. 48–54