The Theory of PSE and the Practice of the ORB(M) Extensible Distributed Object Framework*

Attila Ulbert
Department of General Computer Science
Eötvös Loránd University, Budapest, Hungary
mormota@elte.hu

December 4, 2001

Abstract

Extending an distributed object framework is mostly the duty of its developers. As a result, the user has to wait until the developers implement the desired new service enabling the framework to penetrate into new application fields or to support solving a problem. We think, the user should be allowed to modify or supplement the system with new components, at least on the level of communication subsystem. The pluggable protocol architectures already meet this requirement by enabling the user to implement their own environment or task-specific communication infrastructure.

In this paper we present our flexible approach of extensible Object Request Broker middlewares. The model of *Pluggable Semantic Elements (PSE)* goes further than the pluggable protocol architectures allowing the developer to implement and arbitrarily combine the well-defined functional components of the invocation semantics. Our *ORB(M)* framework implementing the PSE model allows the user to exploit the special characteristics of the environment by extending the library of semantic elements and form new invocation abstractions.

Introduction

Most object oriented frameworks for distributed programming have rather "brittle" architecture. Although they implement the most important distributed interaction modes, such as the transparent synchronous or asynchronous remote method invocations, the user is restricted in influencing or modifying the process of remote invocations. We cannot introduce new distributed interactions solving special problems. As a result of the lack of the flexibility, the unadaptable architecture and the exclusion of the user's assistance the common distributed object frameworks can hardly be used to solve non-traditional problems in such yet unsettled application fields as mobile or GRID [FKT01, ABB+01] computing.

The pluggable protocol architectures [SCH00b] and the Composable Message Semantics (CMS) framework [HOF99] already offer the freedom of implementing new communication mechanisms. While the pluggable protocol architectures allows the user to introduce new communication abstractions, the CMS framework enables the user to decorate the invocation semantics with arbitrary actions. Using Meta Object Protocols (MOPs) [KRB91, FD99] the ORB could offer even more flexibility to the programmers. The MOPs allow the developer to extend the behaviour of the ORB by providing their own meta-classes. Although such systems would offer flexibility and extendibility, they require such a programming environment which supports reflection. This limits their applicability and presumably results lower performance compared to object request brokers based on non-reflexive architectures.

We think, that we should give more freedom to the user allowing them influence all the functional elements of remote method invocations. We recognise, that this level of freedom involves considerable dangers in weakening transparency and interoperability. These dangers however can be limited by extensible testing of the application and formal modelling of the invocation semantics.

According to our model of *Pluggable Semantic Elements* the invocation semantics are described by seven semantic elements and a their ordering. The semantic elements are objects implementing the functionally separated fundamental step of the remote method invocations. Such a fundamental step for example is

*This research has been partially supported by OMFB 01548.

the creation and encoding of network messages or the network communication. Combining these elements and determining their ordering the we can produce invocation semantics with the desired properties.

Our ORB(M) framework [ULB01] implements the PSE model in C++ programming language. Previous measurements proved the framework offers the flexibility of the PSE model without introducing significant overheads in distributed interactions. Moreover, we measured considerable performance improvements using task-specific semantic elements compared to the default interaction mode and an other high-performance CORBA [OMG00] implementation [OMNI]. The system already includes a fairly large library of semantic elements, implementing the common synchronous, asynchronous and oneway invocation, TCP and UDP based oneway and twoway communication, and also includes special elements for transmitting large amount of data or communicating in mobile environments.

The paper is organized as follows. Section 1 presents some special problems the common object request brokers can hardly deal with. Section 2 introduces the model of Pluggable Semantic Elements (PSE). Section 3 presents our ORB(M) system implementing the model of PSE. Section 4 explains how can the users extend the library of semantic elements and introduces the two major groups of task-specific elements. Section 5 presents the dynamic and static customisation of invocation semantics. Section 6 discusses the problem of the applicability and compatibility of the new semantic elements. Finally, Section 7 summarizes the contribution of the paper and gives some concluding remarks.

1 Special Problems and Environments

The common object request brokers and their fundamental concepts were developed to implement remote method invocations. As a requirement, the remote invocations should have to imitate transparently the local invocations, and enable the developer to abstract from the details of network communication and concentrate on the application-level problems. Apart from the handling of communication failures, the twoway synchronous remote invocations almost perfectly meets this requirement. The implementation of this invocation semantics relies on reliable transport protocol, which is TCP in most cases. As the TCP-based twoway remote invocation proved to be suitable to imitate the local invocations in distributed applications, most object request brokers are optimised to a given, mostly reliable, computing environment.

However, the ultimate goal of transparent remote invocations, the relatively friendly computing environment (e.g. reliable transport over fixed networks) and the optimised implementation hardly results extensible or adaptable architectures. The common object request brokers can hardly be used for transmission of large amount of special data required for GRID computing and multimedia, and they can hardly be adapted to such “hostile” environments as the wireless networks and mobile devices [FZ94, HCC99]. Adapting a framework which has been optimised and developed for reliable environments is far from easy. Although, as their widespread use proves, they efficiently solve many problems of distributed object computing, such as the interoperability of distributed objects. Thus, we cannot say that the lack of user-level extensibility essentially limits their application.

The continuous development of distributed object frameworks can offer services solving the well-defined problems of special environments. These services can efficiently be used enabling the framework to penetrate into new areas where they were not intended to be used before. However, such services will never be perfectly suitable for a given application. For example, they can offer GRID specific services and semantics, but not a GRID-based application specific services. Without allowing the user to implement their own application-specific middleware-level abstractions, they cannot exploit the additional information from problem they solve and turn them to their benefit.

Thus, although it’s not vital today, the object request brokers should offer at least partial extensibility to the user. For example, the common frameworks basically use at most a small and fixed set of other invocation semantics besides the “traditional” synchronous interactions. We deem that assuming that all distributed object oriented interactions can be performed using just a fixed number of invocation semantics is very restricting and even penalising, especially in case of multimedia and mobile communication.

Allowing the user to extend the framework with invocation semantics implementing various distributed interactions has many advantages. Firstly, solving the special problems by invocation semantics allows the user to apply high-level abstractions (remote method invocations) in the application layer instead of streams and other low-level solutions. Secondly, implementing special semantics the user can exploit the additional information about the problem and the environment, which for the developers of the distributed frameworks obviously were not available. Thirdly, allowing the user to adapt the framework results higher performance and better applicability in special domains. We have to note however, that the application-specific semantics can endanger the interoperability and can easily result non-standard solutions.

A flexible and extensible yet high performance object request broker cannot abandon the basic concepts of distributed object computing. On the contrary, it has to build upon them making the extension easier. Enabling the user to inject code into the middleware obviously limits the optimisation, as it requires the separation of the fundamental components the user can modify or replace.

In the next section we present or model for invocation semantics based extensible distributed object frameworks.

2 The Model of Pluggable Semantic Elements

The remote invocations are completed according to the invocation semantics bound to the methods participating in distributed interactions. The CORBA [OMG00, OMG] implementations offer the *synchronous*, the *oneway* and the *deferred synchronous* semantics, and *time-independent invocations* [OMG98]. Each invocation semantics can be defined by the transformations they use, and their ordering. These elements of the semantics are well-known in distributed object computing, however their implementation is often tangled inside the middleware in order to result higher performance.

The model of *Pluggable Semantic Elements* separates the transformations and their ordering calling them semantic elements, and allows the user to implement and arbitrarily combine these elements creating various invocation semantics. According to the model, the invocation semantics of a caller or callee method is determined by its *method semantics* meta-object which is responsible for the completion of remote invocation. The method semantics encapsulates the seven fundamental steps of the remote invocation (semantic elements) and the semantic weaver (see Figure 1).

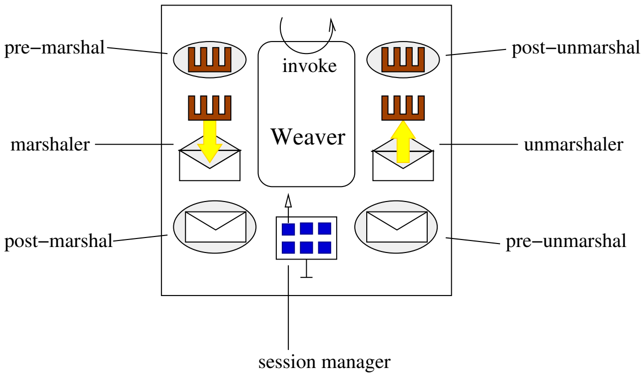

Figure 1: The anatomy of method semantics

The encoding and decoding of network messages are implemented by the *marshaler* and *unmarshaler* semantic elements. They are transformations from a simple parameter format to a transmittable message format, and vice versa. These elements usually have to be generated from the signature of the methods by a compiler, but they can be implemented by the user as well.

The middleware-level infrastructure of network communication is provided by the *session manager* semantic element and the *session* sub-element. The role of the *session manager* is to create the necessary *sessions*. The invocation semantics use the *sessions* to send and receive network messages. In order to enable a session manager to create different sessions, they have a factory parameter. Using the the Factory design pattern [GHJV95] and providing the factory object to the session manager the user can create different communication infrastructures.

The three semantic elements above implement the basic steps of remote invocations. However, there can be problems that should be solved inside the middleware, rather than on the application level. Such a problem can be the synchronisation, the logging and the encryption of invocations. The *pre- and post-marshal/unmarshal interceptors* can implement additional functionalities bound to the remote invocation. The pre-marshal and the post-unmarshal interceptor can modify the parameter object of actual parameters, or take arbitrary course of action based on that parameters. The post-marshal and pre-unmarshal interceptors are similar to the other two interceptors, but they get the marshaled or the received messages.

The *semantic weaver* orchestrates the semantic elements. This special element determines their ordering basically influencing the properties of the invocation semantics. Actually, the semantic weavers implement the remote invocations having certain properties using the seven semantic elements. For example, if it simply marshals the actual parameters, starts a session and sends the message through the network, waits for the response and unmarshals it, then it implements a caller-side synchronous invocation. However, if it spawns a thread which goes through the above steps and passes the results to a callback object, then it implements a caller-side callback-based asynchronous invocation.

Creating *method semantics* containing different semantic elements and weavers results different invocation semantics. The user is allowed to implement and arbitrarily combine semantic elements, and assign the invocation semantics to the methods. This doesn't mean however, that each method should have its own unique, unshared invocation semantics. On the contrary, the method semantics are organised on the object level, and the methods potentially share their semantics. This object level collection of method semantics is the *object semantics*. Figure 2 depicts an object semantics and the object it has been attached to.

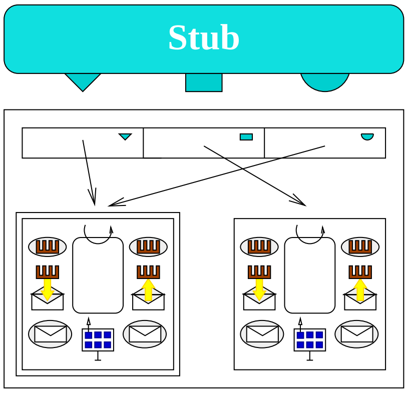

Figure 2: Object semantics as the collection of method semantics

Each server object and client stub has its own especially tailored object semantics. The default semantics of the object specifies the shared semantics of the object's methods. The user is allowed to attach specialised invocation semantics to each method, but if a method has no special semantics, then the default abstraction will be used to carry out the remote invocation. In Figure 2 the client object (the stub of the server object) has three methods. The double framed method semantics is the default semantics, which has been assigned to the first and the third method of the stub. The second method has its own method semantics.

The semantic elements, apart from the semantic weavers, can be shared among different method semantics. This feature is exploited mostly in connection with session managers, as an application rarely requires more than one or two of them.[1] Moreover, this feature enables the user to save more storage capacity by connecting the same marshaler/unmarshaler to methods having the same signature.

The model of *Pluggable Semantic Elements* not only offers flexibility and extendibility, but it can be used to describe the existing distributed object frameworks. The concepts of distributed interactions can be expressed by semantic elements. In order to describe an existing framework we have to determine the abstract semantic elements and their combination. These elements not necessarily exist in a framework we describe, but serve as modelling abstractions.

In the next section we present our *ORB(M)* framework, which directly implements the model of *Pluggable Semantic Elements* and its components.

3 The ORB(M) System

The *ORB(M)* framework is based on the model we presented in the previous Section. The architecture of the framework follows and utilises the concepts of pluggable semantic elements. The implementation preserves the advantages of the model and tries to minimise inevitable negative side-effects of the flexibility, such as the additional overheads. Figure 3 depicts the major components of the *ORB(M)* system.

The complete system will consist of the framework containing the *ORB(M)* core and the library of semantic elements, the *IDL-ISD* compiler and the *Compatibility* checking module. The actual prototype

[1] An average CORBA-based application usually has only one session manager in the terms of PSE model.

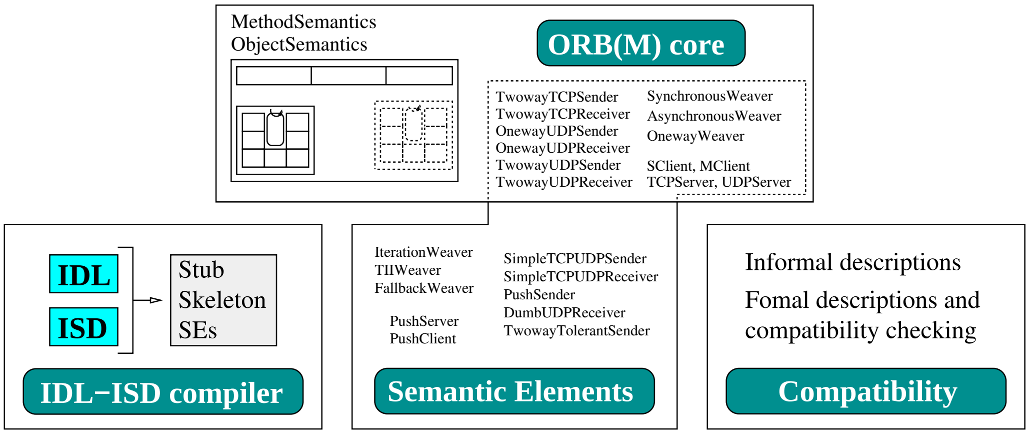

Figure 3: The architecture of the *ORB(M)* system

system consists of the framework and a partially finished compiler, while the module is under development.

The *ORB(M) core* implements the model of PSE. It contains components corresponding to the abstractions of the model, classes and services indispensable for remote invocations. The core also includes parts of the library of *Semantic Elements*. The *core semantic elements* implement the most important communication services as sessions and session managers, and interaction modes as semantic weavers. The core elements implement:

- different policies for session creation and destruction: *SClient, MClient, TCPServer, UDPServer*
- TCP and UDP-based oneway and twoway communication sessions, e.g. a caller-side oneway UDP-based session: *OnewayUDPSender*
- synchronous, oneway and callback-based asynchronous remote interactions: *SynchronousWeaver, OnewayWeaver, AsyncronousWeaver*

The *ORB(M)* also implements a *Life-Cycle* service. This service enables the user to change the remote interface of the client and server objects. We can hide methods of the server object making them remotely unaccessible or order the stub methods to make local calls instead of executing remote invocations.

The distributed object frameworks usually offer more services than a given applications needs. However, personalising the middleware by removing the unused services, like in case of Linux or BSD kernels, we could obtain smaller and more efficient applications. The user can remove the following features of our framework:

- server or client-only code
- interceptors
- replaceable marshalers and unmarshalers
- dynamic customisation of method semantics
- life-cycle service

The library of *Semantic Elements* contains the core and a number of task specific elements. This library can be extended by the user by their own special purpose elements. The library already contains semantic weavers, sessions and session managers for transmission of large amount of data. It also contains elements for disconnected operations in mobile environments.

The *IDL-ISD compiler* (Interface Definition Language - Invocation Semantics Definition) generates the necessary stub and skeleton code, semantic elements and other helper classes based on the interface and (optional) semantics definitions. For interface definitions we used the standard CORBA IDL, for semantic definitions we developed our own language (ISD). The ISD file can be omitted if the users want to use the default invocation semantics (synchronous invocation), or they want to specify the special semantics in run-time.

The *Compatibility* module will be a collection of formal descriptions and techniques enabling the user to examine the properties of the invocation semantics consisting of certain semantic elements. Using the formal techniques the user will be able to check whether the caller and callee-side invocation semantics are compatible, i.e. they can exchange messages and complete remote invocations, and explore the characteristics of distributed interactions. In this stage of development the module only contains the informal descriptions and analysis of the special semantic elements.

4 Extending the Library of Semantic Elements

Before extending the library of semantic elements the developer has to decide what kind of special elements have to be combined in order to obtain the desired invocation semantics solving the given problem. We have note, that the same invocation semantics can be implemented by combining different semantic elements. Moreover, we can produce highly specialised semantic elements and reusable elements as well. Thus, when extending the library the user should decide on implementing reusable elements if it's possible. When the developer identified the necessary new elements, they simply have to inherit from the proper classes representing the seven semantic elements and the weaver.

Usually it's enough to implement the abstract methods of the semantic element classes. There are problems however, which require additional classes and services. For example, we had to provide a service for the *TIIWeaver* in order to effectively implement store-and-forward invocations. Supplementing the library with sessions also requires the implementation of other classes: the factories of the new sessions which must be passed to the session managers.

The special semantic elements often require the assistance of the user weakening the transparency. For instance, in case of multicast session managers the user has to obtain the host address and service port the server expects the control messages. Moreover, the special semantic elements often allows the user to configure or control the in run-time.

In the following three points we give a short overview of the semantic elements already contained by the framework.

4.1 The Core Semantic Elements

The *core semantic elements* provide the infrastructure for the basic distributed interactions. The most important core semantic element is the *SynchronousWeaver* implementing twoway synchronous remote invocations. The weaver executes the semantic elements on their "natural" ordering, from encoding of actual parameters through network communication to the decoding of the response message. The *SynchronousWeaver* is the default semantic weaver of the *ORB(M)* framework.

The *AsynchronousWeaver* implements callback-based asynchronous remote invocations. The weaver applies a pre-spawned working thread-pool completing the remote invocations. It has a parameter specifying the weaver which will be used by the working threads to carry out the invocations. If the user doesn't supply this parameter, the *SynchronousWeaver* will be used. The size of the invocation thread pool can be maximised by the user.

The *OnewayWeaver* implements a simple request-only remote invocation. The weaver executes the steps of the *SynchronousWeaver* up to the request sending then simply closes the communication session and returns to the caller. As a result, it is only applicable when the method doesn't have any return or output parameters.

The core session managers implement the basic infrastructure for session handling. The client-side session managers follow the Connector, while the server-side session managers follow the Acceptor behavioural pattern [SCH00a]. The most often used client-side session manager is the *SClient*. It maintains a session pool in order to minimise the number of resource wasting re-construction of TCP connections. It starts a new session only when the pool doesn't contain and already started one with the requested endpoint. Whenever a new session is started the session manager removes it from the pool, and when a session is stopped, the session manager puts it back to the pool. The *MClient* implements almost the same session-handling policy with the difference that it doesn't remove the started sessions from the pool allowing their shared use.

The functionality of the *TCPServer* server-side session manager is quite simple: it creates a new session when it receives a connection request. A server session usually reads the request header, identifies the target method and activates its invocation semantics; the rest of the invocation including the upcall on the target method will be completed by the semantics. However, the sessions created by the *TCPServer* doesn't have to communicate via the TCP connection accepted by the session manager, for example, they can send response messages via UDP or another TCP connection. The *UDPServer* passes the completely read requests to the sessions it creates, thus they only have to identify the target method's invocation semantics and dispatch the request to it.

The core sessions implement request-only UDP, and twoway TCP and UDP communication. The client-side sessions are called *senders* the server-side sessions are called *receivers*. The *senders* simply serve as a communication infrastructure used by the client-side weavers, while the *receivers* have more complex task. A server-side session has to check whether the target method exists and dispatch the invocation request to its invocation semantics, which will complete the remote invocation and send the response message to the caller using mostly the session received the request. However, as we will see in the following points certain sessions can only be used with a single method, therefore they doesn't need header information to identify the target.

4.2 Transmitting Large Amount of Data

The common object request brokers are often perform poorly when transmitting large amount of data. This, and the lack of group communication infrastructure makes them hard-to-use in multimedia applications. However, there are certain characteristics which can be exploited by special semantic elements and weavers. In library of semantic elements the *IterationWeaver* and the *hybrid sessions* exploit these characteristics, and the *Push* session managers implement group communication.

Exploiting the special invocation patterns of remote methods the user can improve the performance of the applications. Transmitting large amount of data or consequent video frames often result such special patterns. For example, a method returning video frames is presumably invoked in a loop or a thread, and the probability of a single invocation is extremely low. The *IterationWeaver* semantic weaver implements a single-request infinite-response twoway invocation, i.e. the client-side weaver sends only one request message, while the server-side weaver executes "infinite" upcalls and returns the responses. Actually the server-side weaver stops working when the client closes the communication session, which happens when the caller stops calling on the remote method. This strategy is only applicable when the client indeed invokes the remote method in rapid succession, otherwise the negative "side-effects" of the weaver will be much expressed. Such side-effect is the limited request sending, i.e. one cannot send new invocation requests until the underlying session is closed. An other side-effect that the response we get doesn't always reflect the current state of the object.

The *hybrid sessions* are based on the realisation that certain distributed interactions sensible of high delays could allow the loss of response messages. The *SimpleTCPUDPSender* (client-side) and the *SimpleTCPUDPReceiver* (server-side) session transmit the requests via TCP and the responses via UDP. Using the *hybrid* sessions with *IterationWeaver* we can experience even delays *30%* lower than in case of the default (synchronous) interaction mode.

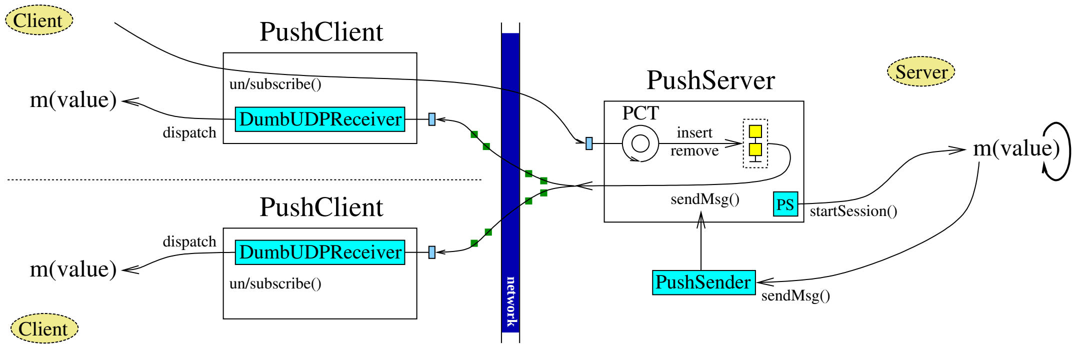

Figure 4: Multicast remote methods invocations based on *Push* session managers

The *Push* session managers implement UDP-based multicast communication (see Figure 4). When the server application invokes on a method (m) using this session manager the invocation request is sent to all the clients subscribed to the given *PushServer*. Following from this kind of reversed logics of multicast invocations the server contains the caller method and the clients contain the callee methods. The invocation semantics of the callee methods have to use the *PushClient* session manager. The user has to explicitly define the network address and the number of the service port the *PushServer* expects the control (un/subscribe) messages. The user can subscribe and unsubscribe the client in run-time.

There is an other simple UDP-based session implementing fast network communication. The *DumbUDPReceiver* enables the user to receive headerless messages. In order to use this session the user has to supply it the reference of the method is has been bound.

4.2.1 The Corpus parametrised class

As we have seen in the case of *push* session managers sometimes the developer has to apply a kind of reversed logics in their applications. Such reversed logics leads to mixed client and server codes, simultaneous use of stubs and skeletons. The *Corpus* class template encapsulates the functionality of the stubs and skeletons.

Actually, the *Corpus* is a stub, which has a skeleton implementing server functionality and a local object (these are the two class and one object parameters of the class template). The methods of the *Corpus* can play the following roles:

- *client* method, invoking a remote method and returning the results of the remote invocation
- *remote* method, serving the remote clients using the local object supplied by the user
- *local* method, calling on the proper method of the local object

The implementation of the *Corpus* heavily relies on the *Life-Cycle* service. The *Corpus* inherits the methods of the stub and implements its own control methods allowing the user to determine the roles of each method.

4.3 Semantic Elements for Mobile Computing

In wireless networks and mobile environments the user often have to deal with frequent disconnections and high delays. We have implemented three semantic elements in order to weaken the undesired effects of mobility: a fault-tolerant session, a semantic weaver implementing store-and-forward invocation, and an other weaver enabling the system to forward the invocation request to local proxy objects.

The sessions we have already implemented in our ORB(M) framework are expecting reliable transport. The *TwowayTolerantSender* (client-side) session is able to build the transport connection in deferred manner, i.e. if the connection cannot be created when the session is started, it does not throw an exception, but builds it when it becomes possible. This session is able to continue request sending and response receiving after previous communication failures. Moreover, the session does not need buffering, it simply appends the already read message with the new fragments.

The *store-and-forward* communication strategy implemented by (client-side) *TIIWeaver* (Time-independent Invocation Weaver) and the `TIIService` fits better to mobile communication than the *thread pool model* applied by the *AsynchronousWeaver*. The remote invocations are carried out by the service, the weaver simply passes the necessary information (callback object, semantic elements, etc.) to it. If the service is unable to complete the remote invocation due some communication failure it stores the request and tries to forward it later. In order to save storage capacity the user can limit the number of pending requests.

When a remote invocation cannot be completed because of the disconnection we can use local objects with simplified functionality to obtain the data we want. This can be done by implementing a local proxy object, put the remote invocations into a *try* block and calling the proper method of the proxy object in the *catch* block. This solution however makes the application code more complex and less efficient as the communication exceptions have to "travel" up to the application level. The *FallbackWeaver* applies the above strategy inside the middleware, which makes it more efficient resulting delays about *30%* lower than the application-level operations. The weaver contains a similar *try-catch* block and calls on the proxy object supplied by the user when the weaver is created.

Finally, we have to mention that modifying the un/marshaling semantic elements can also improve the performance of wireless and multimedia applications. Compressing certain parameters can save valuable bandwidths and reduce the otherwise high delays. The compression can be lossless or even depending on the contents of the parameters.

5 Customisation of Invocation Semantics

The default semantic elements and invocation semantics provided by the compiler usually prove to be suitable, and there are just a few semantics requiring modifications. The customisation allows the user to replace or specify the invocation semantics of these methods. The *dynamic* customisation enables the user to arbitrarily change the invocation semantics tailored to methods in run-time, the *static* customisation specifies the semantics in compile-time.

5.1 Dynamic Customisation

Using customised invocation semantics potentially weakens transparency, especially in case of dynamic operations. The user have to take care of the special invocation semantics he assigns to various objects, and often deal with the details of semantic element creation. However, the performance improvement resulted by non-default special semantics often counteracts these disadvantages.

The code-snipped below depicts a `Video` interface, which will be used by the clients to receive video frames. Method `nextFrame()` simply returns the next video frame of a stream. Suppose, that the user wants to apply the *IterationWeaver* and the *SimpleTCPUDP* sessions.

```
typedef char Frame[32768]; /* 32K */
interface Video {
  Frame nextFrame();
  ...
};
```

There are two ways to change the invocation semantics of a method: you can create completely new semantics, or modify the (default) semantics already attached to the method. In our case, we will use the the second method as we want to apply only minor changes to the semantics, i.e. replace two semantic elements. In order to attach customised semantics to a method we have to go through the following steps: (1) obtain the semantics already attached to the method or create a new semantics; (2) create the semantic elements and assign them to the semantics; (3) attached the new semantics to the method.

The code-snippet below depicts the process of client-side dynamic customisation:

```
... // obtain the object reference "ref"

Video_stub video(ref); // create the stub object

// (1) obtain the semantics of method "nextFrame"
MethodSemantics ms = StubHelper::getMethodSemantics(video, "nextFrame");
// (2) replace the old weaver with "IterationWeaver"
ms.setSemanticWeaver(new IterationWeaver(10, 200000, 200000)); // timeout == 0.2 second
// (2) replace the old session manager
ms.semanticElements()->setSessionManager(new SClient(new SimpleTCPUDPSenderFactory(), 5));
// (3) attach the customised semantics
StubHelper::setMethodSemantics(video, &ms, "nextFrame");

while (true) {
  Frame f = nextFrame(); // call method "nextFrame" as usual
  ...                    // put frame "f" onto the screen
}
```

We have to use helper classes to access and assign invocation semantics. A method is identified by its name. The semantic elements can be replaced by using the simple *set* methods of class *MethodSemantics* and *SemanticElements*. After the user attached the modified semantics, the method can be invoked as usual. The server-side operations are similar, the only major difference that we have to create a local object as well, which will be used by the *skeleton* as the server object, and if we don't want the clients to use our local object we have to hide its methods during the semantic operations (see the code-fragment below).

```
Video_impl obj;                     // the local object
Video_skel video(obj, NULL, false); // the skeleton, won't be exported (see "false")

// (1) obtain the semantics of method "nextFrame"
MethodSemantics ms = SkeletonHelper::getMethodSemantics(video, "nextFrame");
... // (2) assign the new semantic weaver and session manager
// (3) assign the new semantics
SkeletonHelper::setMethodSemantics(mails, &ms, "nextFrame");

SkeletonHelper::exhibit(video); // export the object (make it remotely accessible)
```

5.2 Static Customisation

The invocation semantics the methods use are mostly fixed in run-time, and this makes dynamic customisation unnecessary or even undesirable. We have already seen that replacing the default semantics in run-time requires the user to mix the application code with semantic definitions. Moreover, changing the invocation semantics in run-time can make the application unpredictable.

Introducing the compile-time static customisation has several benefits. First of all, the user doesn't have to mix the application code with semantic definitions. Secondly, the definitions are encapsulated in a separate *Invocation Semantics Definition (ISD)* file. Thirdly, the application is more predictable as the methods obtain their semantics when the stubs and skeletons are created and the semantics doesn't have to be changed afterwards. Fourthly, we obtain more efficient code.

Apart from the above advantages of the static customisation, it is simpler than the dynamic operations: the user simply identifies and creates the non-default semantic elements and completes some additional operations[2] if necessary. As there is no semantics bound to the object when it's being created the user doesn't have to obtain the semantics. The user doesn't have to deal with the assignment of semantic elements and the invocation semantics as well, because this will be done by the *IDL-ISD* compiler.

The *ISD* file has three parts: the first two contain C++ code, while the third part contains the semantic definitions. The compiler copies the first part into the skeleton, the second part into the stub file verbatim. The *semantic definition blocks* of the third part are woven into the stub and skeleton codes [AOP, KIC96]. The code-snippet below depicts the static invocation semantics definition of interface `Video`:

```
semantics Video {
  client Video[smgr: s] { // caller-side default semantics
    SessionManager* s = new SClient(new TwowayUDPSenderFactory());
  }
  server Video[smgr: s] { // callee-side default semantics
    SessionManager* s = new UDPServer(new TwowayUDPReceiverFactory());
  }

  client nextFrame[smgr: s, sw: w] { // caller-side semantics of method "nextFrame"
    SemanticWeaver* w = new InterationWeaver(10, 200000, 200000);
    SessionManager* s = new SClient(new SimpleTCPUDPSenderFactory(), 5);
  }
  server nextFrame[smgr: s, sw: w] { // callee-side semantics of method "nextFrame"
    SemanticWeaver* w = new IterationWeaver();
    SessionManager* s = new TCPServer(new SimpleTCPUDPReceiverFactory(), 0, 5);
  }

  ...
};
```

In order to specify the non-default invocation semantics of the methods of an interface, the user have to write a *semantics* block. The block consists of the semantic definitions of the specialised methods and the default invocation semantics. The semantic definition of a given method is identified by the method name, while the definition of the default semantics is identified by the name of the interface. The identifier

[2] For example, read configuration information from a file and set-up the semantic elements accordingly.

has to be preceded by keyword *client* in case of caller-side definitions, and keyword *server* in case of callee-side definitions. The identifier is followed by the semantic elements we want to customise, listed in square brackets. The customised semantic elements are identified by labels [3] separated by a colon from their actual value, which is a variable of the *semantic definition block*. The list of customised semantic elements is followed by the *semantic definition block* in braces containing the C++ code, which initialises the customised elements. In our example, the default (caller and callee-side) invocation semantics of the interface will apply twoway UDP-based communication, while the semantics of method `nextFrame()` will use the *IterationWeaver* and hybrid (TCP and UDP combined) sessions.

6 Applicability Assessments and the Question of Compatibility

Whenever the developer supplement the library of semantic elements they have to assess the applicability and the characteristics of the new element. Without this assessment the user cannot decide whether the new element is suitable for solving a given problem or not. Moreover, the user have to know the elements the new element cannot cooperate with and form an invocation semantics. This characterisation can be achieved by using formal methods, however in it often requires such an additional effort from the developer, that just a few of the is ready to make.

In order to assess the applicability of semantic elements we have to state the most important guidelines of analysis. These guidelines have to be considered whenever the developer make their assessments. The most important elements of the informal descriptions and applicability assessments:

- the circle of other elements the given element can *cooperate* with and form an invocation semantics
- the *effect* the given element produce on the others in an invocation semantics, and vice versa
- the *transparency* level; i.e. whether the user have to deal with some additional operation when they are using the given element, and whether this operation has an effect on the compile- or run-time customisation of the invocation semantics
- the *performance* of the given element, which can be measured by some features of its output, the number it can be executed in a given time period, or in any other means; note that the performance of the given semantic element largely determines its applicability

Naturally, the above points cannot cover all the important features a semantic element can have, however without these the characterisation is obviously incomplete. Moreover, these assessments are bound to a stage of development and a set of semantic elements. Thus, the developer have to refresh at least his own the assessments whenever he extends the library of semantic elements.

Even in case of well described and analysed semantic elements the user cannot disregard the run-time testing of customised invocation semantics. Although the semantic elements generated by the *IDL* compiler are compatible with one another, the system cannot ensure compatibility in case of specialised semantics. The interoperability and compatibility testing can be supported by the formal modelling of the system and the semantic elements.

6.1 The Role of Formal Modelling and Compatibility Checking

The properties of an invocation semantics is determined by the properties of its semantic elements and their interaction. As we have already stated, after implementing a new semantic element we have to analyse its features and give at least an informal description. However, we need formal methods to describe exactly the properties and interactions of the semantics. We usually search for the answers to the following questions:

- ☐ What are the properties of a given invocation semantics formed by the given semantic elements?
- ☐ Are the caller and callee-side invocation semantics of a given method able to complete remote invocations? (compatibility)
- ☐ What are the characteristics of the interaction between two (compatible) invocation semantics?

[3] Label *smgr* identifies the session manager; *pm* identifies the pre-marshal, *tm* the post-marshal, *pu* the pre-unmarshal, *tu* the post-unmarshal interceptors; *m* identifies the marshaler, *u* the unmarshaler; *sw* refers to the semantic weaver.

Based on informal descriptions, the answers would be exact at most by chance. Thus the *Compatibility* module should include formal descriptions of semantic elements and weavers, and techniques for deciding the above questions.

We have already provided an abstract formal model of the *JavaCMS* system based on evolving algebras [GUR88] and described various invocation semantics such as the *Time-independent Invocation (TII)* in Propositional Temporal Logics (PTL) [KRO87, MP79, HU01]. The formal model gave a solid ground of the system and offered better insight to the users. The PTL-based abstract description introduced the basic properties of the *synchronous, asynchronous* and *TII* invocation semantics. The description gave a clear view of the invocation semantics implemented in *JavaCMS*. In our *ORB(M)* system we try to give a similar, but more easy to understand description of invocation semantics in UNITY [CM88], and present methods helping the developers to decide the question of compatibility.

7 Conclusions and Future Work

Limiting the user in extending the ORB middleware is often penalising. The inextensible systems can hardly be adapted to the actual computing environment and the application, offer lower performance when using in non-traditional environments and limits the structuring of the application. If a problem is not solved by the middleware itself, the user is forced to deal with it on the application level independently of its nature. This results more complex application code and unstructured horizontal rather than hierarchical architectures.

We believe that the mode of distribution interaction should, like many other programming aspects, be represented by an extensible class library [LEA97], and not be hard coded in the language. Our *Pluggable Semantic Elements* model offers the freedom of "middleware programming". In our *ORB(M)* framework implementing the model the user is allowed to implement and combine arbitrary semantic elements creating various different distributed interaction modes. The framework however ensures the transparent synchronous invocation as its default interaction mode. The library of semantic elements already contains abstractions supporting the transmission of large amount of data or working in mobile environments. Using the special semantic elements we experienced considerable performance improvement, even in comparison with one of the fastest CORBA implementation [OMNI].

The user can replace the invocation semantics in run-time or specify them in compile-time. The dynamic operations should be avoided if we don't have to change the semantics of a method more than once in run-time. The customised invocation semantics raise the question of compatibility. Using special abstractions on the client and server-side we can loose the interoperability. Moreover, the circle of semantic elements a given element can cooperate with is also limited.

Thus, besides the development of *IDL-ISD compiler* we will focus on the question of formal modelling and compatibility. There are several formal languages and models which have already proved their applicability describing real problems and systems [KRO87, MP79, CM88, MR98, MR99]. This formal model will provide a solid foundation for the *ORB(M)* system, and may be able to describe other distributed object frameworks as well.

References

[ABB+01] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke, *Data Management and Transfer in High-Performance Computational Grid Environments*, Parallel Computing, 2001.

[AOP] Homepage of the Aspect-Oriented Programming Project, XEROX Palo Alto Research Center (XEROX PARC), Palo Alto, California, `http://www.parc.xerox.com/aop`

[CM88] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[FD99] Ira R. Forman and Scott H. Danforth, *Putting Metaclasses to Work*, Addison-Wesley, 1999.

[FKT01] I. Foster, C. Kesselman, S. Tuecke, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, Intl. J. Supercomputer Applications, 15(3), 2001.

[FZ94] George H. Forman, John Zahorjan, *The Challanges of Mobile Computing*, IEEE Computer 27, pp. 38-47, April 1994.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[GUR88] Yuri Gurevich. Logic and the challenge of computer science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1-57. Computer Science Press, 1988.

[HCC99] Mads Haahr, Raymond Cunningham and Vinny Cahill, *Supporting CORBA Applications in a Mobile Environment*, Proceedings of the 5th International Conference on Mobile Computing and Networking (MobiCom'99), pp. 36-47, Aug. 1999.

[HOF99] Markus Hof, *Composable Message Semantics in Object-Oriented Programming Languages*, Trauner, ISBN 3 85487 118 X, 1999.

[HU01] Markus Hof, Attila Ulbert, *Time Independent Invocation in JavaCMS*, Periodica Polytechnica Ser. Electrical Engineering, 2001.

[KIC96] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. Videria, C. Maeda Lopes, and A. Mendhekar, *Aspect-Oriented Programming ACM Computing Surveys*, 28(4es), 1996.

[KRB91] G. Kiczales, J. des Rivières and D. Bobrow, *The Art of the Metaobject Protocol*, The MIT Press, 1991.

[KRO87] F. Kröger, *Temporal Logic of Programs*, Springer Verlag, Berlin Heidelberg, 1987.

[LEA97] D. Lea, *Concurrent Programming in Java.* Addison-Wesley, 1997.

[MP79] Z. Manna, A. Pnuelli, *The Modal Logic of Programs*, LNCS 71, pp. 385-409, 1979.

[MR98] Peter J. Mccann and Gruia-Catalin Roman, *Compositional programming abstractions for mobile computing* IEEE Transactions on Software Engineering 24, 2 (Feb.), pp. 97-110, 1997.

[MR99] Peter J. Mccann and Gruia-Catalin Roman, *Modeling Mobile IP in Mobile UNITY*, ACM Transactions on Software Engineering and Methodology, Vol. 8, No. 2, pp. 115-146, April 1999.

[OMG] The Homepage of Object Management Group, `http://www.omg.org`

[OMG98] Object Management Group, *CORBA Messaging Specification*, OMG Document orbos/98-05-05yes ed., May 1998.

[OMG00] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, version: 2.4, October 1, 2000.

[OMNI] The Hompage of omniORB: `http://www.uk.research.att.com/omniORB/`, AT&T Laboratories - Cambridge

[SCH00a] Douglas C. Schmidt, *Pattern-Oriented Software Architecture: Pattern for Cuncurrent and Networked Objects*, Wiley & Sons, ISBN 0-471-60695-2, 2000.

[SCH00b] Douglas C. Schmidt, Carlos O'Ryan, Ossama Othman, Fred Kuhns and Jeff Parsons, *Design Patterns in Communications, Applying Patterns to Develop Pluggable Protocols Framework for ORB Middleware*, (Linda Rising ed.), Cambridge University Press, 2000.

[ULB01] Attila Ulbert, *Pluggable semantic elements and invocation semantics*, ECOOP'2001, PhD Workshop, `http://www.st.informatik.tu-dramstadt.de/phdws`, Budapest, Hungary, June, 2001.