Interactions in Distributed Programs Based on Decoupled Communications[1]

Andreas Polze

Institut für Informatik
Humboldt-Universität zu Berlin
Lindenstr. 54a
10117 Berlin
Germany
e-mail: apolze@informatik.hu-berlin.de

Abstract

Distributed applications can be designed and structured with respect to different communication paradigms. Here we investigate a transformation from message-based and imperative communication style to decoupled communication. We refer to the *Object Space* approach as a platform for distributed computation which integrates decoupled communication with object-oriented essentials. Relying on *Object Space* we develop classes which provide constructs for message-based and imperative communication. As a side effect these classes enhance the first two communication styles with new aspects such as dynamic reconfiguration and load balancing.

We develop a generic `channel` class which provides message-based communication through member functions `send` and `receive` to the programmer. Typed communication channels may be defined as subclasses derived from `channel`.

The concept of active objects is a means to combine concurrent and object-oriented programming. We show how active objects in C++ may be implemented on top of *Object Space*. We develop the class `Concurrency` as a base class for active objects. Communication between clients and servers as implemented by active objects follows the imperative paradigm. Thus we demonstrate how imperative communication may be mapped onto decoupled communication.

1. Introduction

Distributed applications may be implemented using different paradigms of communication and interaction. Message-based, imperative, and decoupled communication are equivalent in the sense that a program written in one style can be rewritten in any of the others. Each approach, however, has its own domain of application where it is better suited for solving problems than others.

Decoupled communication seems to fit very well into the context of open distributed systems. Our focus will be on its integration with object-oriented mechanisms, which allows for a higher level of abstraction by encapsulating communication protocols used between components of a distributed application in classes.

Using two distributed algorithms as examples we show how message-based and imperative communication may be mapped onto decoupled communication. This is done by developing special classes which support a particular style of communication through member functions. The implementation of those classes employs a series of *Object Space* operations.

In message-passing programs, processes share *channel*s. A channel is an abstraction of a physical communication network in that it provides a communication path between processes. Channels are accessed by means of two primitives: *send* and *receive*. With asynchronous message passing, channels have conceptually unbounded capacity. Therefore the *send* primitive does not cause a process to block. A process may eventually block

[1] appeared in the Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS) USA'94, Santa Barbara, pp. 81-91, August 1994.

when performing a *receive* operation and thereby synchronize with a communication partner. In our discussion we consider channels to be global to processes.

The notion of imperative communication covers concepts like *remote procedure call* (RPC) and *rendezvous*. Both concepts combine aspects of monitors [Hoare 74] and synchronous message passing. As with monitors, a process exports operations and the operations are invoked by a *call* statement. As with synchronous message passing, execution of a call causes the calling process to delay until results of the invoked operation are obtained. However, operations not returning any result may be handled asynchronously, the caller does not block in this special case. An invocation of an operation may be handled in two different ways: either a new process is created to execute the operation (RPC) or a *rendezvous* with an existing process takes place.

Decoupled communication or *generative communication* [Gelernter 85] is similar to asynchronous message passing. Processes share a single communication channel, called *tuple space*. Associative naming is used to distinguish different kinds of tuples (messages) stored in the *tuple space*. Generative communication is decoupled in time and space: even at runtime the originator of a message does not (necessarily) know its receiver nor does the converse hold.

The *Object Space* approach integrates object-oriented programming paradigms such as inheritance and data encapsulation with decoupled communication between components of a distributed application. Within our model processes may exchange arbitrary objects. The associative addressing scheme takes inheritance into account when it attempts to match two objects. Application-specific *matching* functions allow for complex communication steps (e.g. access to multiple objects) to take place within one single *Object Space* operation. The *Object Space Language* (*OSL*) is used to express communication and synchronization constructs. It has been embedded into the sequential object-oriented language C++.

In the following section we briefly outline the main ideas of the *Object Space* approach. Later we discuss a distributed sorting algorithm based on asynchronous message passing. We develop classes which allow expression of this communication style with a series of *Object Space* operations. An I/O system for distributed applications based on active objects serves as an example for client/server interactions using imperative communication. We present the concept of active objects as a means for transforming imperative communication into *Object Space* communication. Finally we discuss advantages of our solutions using *Object Space*.

The examples presented have been implemented in C++ on top of our prototype version of *Object Space* within a network of UNIX workstations.

2. The *Object Space* Approach

The *Object Space* approach for distributed programming [Polze 93a][Polze 93b] integrates a Linda-like communication style with object-oriented mechanisms such as inheritance, data encapsulation and polymorphism. Furthermore, *Object Space* introduces the concept of application-specific *matching* functions. *Object Space* constitutes a distributed associatively addressed memory. Components of a distributed application may access this memory and store or retrieve objects.

Four operations are available within *Object Space*:

- **`out`** writes an object into the *Object Space*.
- **`in`** and **`rd`** take a template (a special object) as an argument. Both operations retrieve a matching object from the *Object Space* and store its values into the template; they block until a matching object is found. In addition, **`in`** removes the object from *Object Space*. An object matches a template depending on its class and on the values of its data components. If the template has less data components than a retrieved object (i.e., object's class is derived from template's class), extra components are silently discarded.
- **`eval`** creates a new UNIX process either locally or remotely. Either it carries the command line arguments for a remote process or it carries address and arguments of a function, which is to be executed locally, as parameters.

Objects within *Object Space* are represented as an aggregate of class-specific type information and its data components. Thus within *Object Space* an object has no identity. It is not possible to activate member functions for an object stored in *Object Space*. Those objects rather serve as units of communication. They may be stored into or retrieved from *Object Space* via operations **`out`**, **`rd`** and **`in`**. These operations copy an *Object Space* object onto an ordinary C++ object or vice versa. Since operations **`out`**, **`rd`** and **`in`** deal with an object as a whole only, *Object Space* retains the protection mechanisms found in C++. Therefore, communication details may be hidden by construction of appropriate C++ classes, which may then provide secure interfaces to its clients. C++ classes may be used for defining communication protocols. These protocols describe the interactions between components of a distributed application. Previously defined protocols may be extended by use of inheritance. This view gives a new level of abstraction to the programmer of distributed applications. A prototype version of *Object Space* has been implemented in C++ under UNIX.

3. Message Passing via *Object Space*

As an example for a distributed program relying on message passing we discuss an implementation of a sorting algorithm. The problem consists of sorting `n` integer numbers in ascending order. The simplest solution to our problem is a filter program which reads all input data, subsequently sorts the data using one of the well known algorithms and writes all output.

One minor problem is to determine when all of the input has been read and the sorting process may start. This can be accomplished by providing the number of items to be sorted as an additional first input element or by using a special sentinel in the data stream.

Another solution for our sorting problem uses several processes instead of only one. These processes may work in parallel. Each of them receives two sorted lists of input data and creates a sorted output list by merging the former two. Each list of data is terminated by a well-defined sentinel. By composition of those merge processes the sorting network shown in Fig. 1 may be obtained:

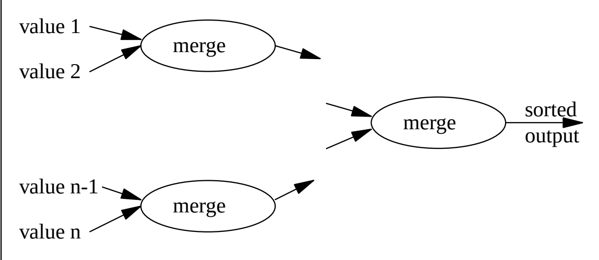


Fig. 1*: Sorting network*

We describe the algorithm of a merge-process using asynchronous message passing with *send/receive* through channels. Later we discuss how this style of communication may be translated into sequences of *Object Space* operations. We define a class `channel` which encapsulates those operations. The semantics of `send` and `receive` is provided through the functional interface of class `channel`. Communication occurs via typed messages. For each data type used in messages a special class has to be derived from `channel`.

In our example integer numbers have to be sorted by a function `merge`. This function accepts identifiers for two input channels and an output channel as parameters. Access to those channels is provided by instances of class `int_channel`. The function `merge` is implemented in C++. The special value `EOS` denotes the end of a data stream.

```
void merge(char* in_chan1,
      char* in_chan2, char* out_chan){
  int_channel in1(in_chan1);
  int_channel in2(in_chan2);
  int_channel out(out_chan);

  int v1 = in1.receive();
  int v2 = in2.receive();
  do {
    if ((v1!=EOS) && (v2!=EOS)){
      if (v1<v2){
        out.send(v1); v1=in1.receive();
      } else {
        out.send(v2); v2=in2.receive();
      }
    } else if ((v1!=EOS) && (v2==EOS)){
        out.send(v1); v1=in1.receive();
    } else if ((v1==EOS) && (v2!=EOS)){
        out.send(v2); v2=in2.receive();
      }
    } while ((v1!=EOS) || (v2!=EOS));
    out.send(EOS);
  }
```

Fig. 2: *Implementation of function merge*

Function `merge` contains a main loop which is executed as long as input data is available. After processing all input the sentinel element `EOS` is written into the output channel and function `merge` terminates. To construct a sorting network several processes are needed, each of them performing one *merge* operation. If the number of input values is a power of two, by proper connection of those processes through channels the sorting network shown in Fig. 1 can be constructed.

The Transformation of *send/receive* Operations

Before we continue with the discussion on how to connect input and output channels of `merge` processes, we describe how asynchronous communication with *send/receive* operations may be implemented based on *Object Space*. A channel is interpreted as a set of messages (objects) stored in *Object Space*. Each of these objects has three data components: a channel identifier, an ordinal number and a description. They are realized as instances of class `channel`. Within this class an ordering relation among objects in a channel is implemented. Two special objects describe the first and the last object (message) within a channel. Classes like `int_channel` may be derived from class `channel`. Objects of class `int_channel` extend those of class `channel` by an additional integer component.

An arbitrary number of processes may perform *send* or *receive* operations on a particular channel simultaneously. They synchronize when accessing the channel. Objects of class `channel` and of classes derived from `channel` may rather be understood as a handle to a channel than as the channel itself. Their component functions *send/receive* may be used to write messages to a channel or to read them. To create the two special objects belonging to a channel as mentioned above, the function `init_channel()` has to be called. `remove_channel()` is the converse operation. A channel is identified by a name string. In Fig. 3 the classes `channel` and `int_channel` are shown.

```
class channel {
    char * chan_id;
    int ord;
    char * desc;
    virtual channel * mk_templ(int ord);
public:
    channel(char * id);

    init_channel();
    remove_channel();
    void send();
    channel * receive();
};

class int_channel: public channel {
    int val;
    virtual channel * mk_templ(int ord);
public:
    int_channel(char * id);

    void send(int val);
    int receive();
};
```

Fig. 3: *The classes channel and int_channel.*

Two special objects describing the first and the last message stored in a channel belong to each channel. The component `desc` of those special objects has either the value `first` or `last`. In contrast, the component `desc` has the value `NULL` for each of the objects stored as messages in a channel.

Now we describe in greater detail the operations `send()` and `receive()` residing in class `channel`. `send()` first performs *Object Space* operation **`in`** to exclusively read the object with description last. In what follows we will call this object *last*-object. Next, operation `send()` increments the ordinal number of the *last*-object and writes a data object (a message) with the same ordinal number into *Object Space* using operation **`out`**. Afterwards the *last*-object with incremented ordinal number is written back into *Object Space* using **`out`**. Within our example class `channel` ordinal numbers are represented as integers. All increment operations are handled modulo MaxInt, where MaxInt is the maximum integer value in our implementation. This can cause problems if the capacity of a channel really has to be unbounded.

In this case we would introduce a special class `ordinal_number` with an infinite number of different values. Instances of this class could be used to denote messages in a channel. Their values could be represented within dynamic character strings.

Member function `receive()` applies operation **`in`** to obtain exclusively the object with description `first`, subsequently called *first*-object. Next, `receive()` performs *Object Space* operation **`in`** to obtain the data object (a message) with the same ordinal number as the one stored in the *first*-object. During that process the member function `receive()` may eventually block — in this case no message is contained within the channel. Finally the ordinal number of the *first*-object is incremented. The object is written back into *Object Space* using **`out`**.

Both *first*- and *last*-object may either be stored in *Object Space* or owned by exactly one process. Processes which deal with a channel simultaneously synchronize when accessing the *first*- or *last*-objects. As channels are completely represented within *Object Space* they are accessible from processes on different network nodes.

To allow for reading a data object — a message within a channel — with member function `receive()` a template object is needed for *Object Space* operation **`in`**. A pointer to a template object of class `channel` must be obtained by calling the virtual function `mk_templ()` of class `channel`. Derived classes may redefine this function to return a pointer to one of *their* instances. Consequently, within class `int_channel` a data object is read which contains an integer component `val` — the real message.

Operations `send()` and `receive()` of class `channel` as shown in Fig. 4 have no parameters. Instead of moving actual data into or out of a channel, these functions implement all the bookkeeping necessary to represent a channel as a set of objects within *Object Space*. The special objects describing the contents of a channel (*first*- and *last*-objects) are maintained correctly. The synchronization between processes trying to access a channel concurrently is also handled correctly within functions `send()` and `receive()` in Fig. 4.

```
void send() {
  channel c( chan_id );

  c.formal( ord );
  c.desc = "last";
  c.in();
  c.ord += 1;
  ord = c.ord;
  out();
  c.out();
}
```

```
channel * receive() {
  channel c( chan_id );

  c.formal( ord );
  c.desc = "first";
  c.in();
  c.ord += 1;
  channel *d = mk_templ( c.ord );
  d->in();
  c.out();
  return d;
}
```

```
channel * mk_templ( int _ord ) {
  channel * p = new channel( chan_id );

  p->ord = _ord;
  p->desc = NULL;
  return p;
}
```

Fig. 4: *send(), receive() and mk_templ() in class channel*

Function `mk_templ()` is used to generate an instance of class `channel` or one of its subclasses as a template for *Object Space* operation **`in`**. This function is dynamically bound (*virtual*), it may be redefined in derived classes. Thus, following the rules of data polymorphism in C++ and *Object Space*, an instance of a derived class may be read or written instead of a `channel` object.

In Fig. 5 we present the versions of `send` and `receive` associated with `int_channel`. These operations are implemented to allow for passing integers as messages. Class `int_channel` demonstrates how language constructs for asynchronous message passing may be implemented within a class based on *Object Space* communication.

```
void send( int _val ) {
  val = _val;
  channel::send();
}
```

```
int receive() {
  int_channel * p = channel::receive();
  int val = p->val;
  delete p;
  return val;
}
```

```
channel * mk_templ( int _ord ) {
  int_channel * p;

  p = new int_channel( chan_id );
  p->ord = _ord;
  p->desc = NULL;
  formal( p->val );
  return p;
}
```

Fig. 5: *send(), receive(), mk_templ() in class int_channel*

The versions of `send()` and `receive()` as shown in Fig. 5 rely on their counterparts in class `channel`. But they also deal with the value of the integer component `val`. Function `mk_templ()` in Fig. 5 yields a pointer to an instance of class `int_channel`, thus ensuring that an `int_channel` object is read within function `receive()` in class `channel`.

Configuration of the Sorting Program

To activate our sorting network it is necessary to create a number of channels and merge processes within *Object Space*. The channels' names have to be provided to the processes. Since all processes are identical — each of them executes an incarnation of function `merge()` — it is irrelevant which channels a particular process uses as input channels. It merely has to be ensured that no cycles appear in our sorting network. Let us assume that channels are identified by integer numbers. They may also be represented as objects within *Object Space*, thus being accessible to all processes.

A main process creates all communication channels via `init_channel()`. Afterwards each process reads an identifier for its output channel. Let the number of input values be $n = 2^k$ for natural k. Then a perfect sorting network consists of $n - 1$ merge processes communicating through $2 * n - 1$ channels. Let the identifier i, $1 \le i \le n - 1$ describe output channels. As an example let us consider the output channel with identifier i. Corresponding input channels carry the identifiers `2*i` and `2*i+1`. The main process fills the channels $n \ldots 2 * n - 1$ with input data and finally receives sorted output data from 1.

With our example we have discussed not only a (pretty complex) distributed sorting program but have described how message passing with *send/receive* may be realized on top of *Object Space*. The results are typed channels which may be (re-)configured dynamically. Furthermore those channels may be accessed by an arbitrary number of processes simultaneously. The constructs *send* and *receive* are provided through a class.

4. Imperative Communication

We demonstrate an approach for concurrent programming based on *Object Space* as a way to implement imperative communication between clients and servers. Active objects combine the concepts of object and process. The behavior of an active object is implemented by a separate *Object Space* process. Active objects are instances of a class derived from class `Concurrency`. Then concurrency within a distributed application may be seen as creation of several active objects and their interactions.

Active objects implement the client/server model in distributed environments. In our approach active objects are implemented by at least two C++ objects. A *server* object implements the functionality of the active object within a separate *Object Space* process. This object accepts requests and processes them by invoking of private component functions. Since the *server* object resides in its own address space, it is necessary to introduce a means for accessing the *server* object from different address spaces. This can be achieved by a *proxy* object which represents a handle to the *server* from within a client's process. A call to a component function in a *proxy* object results in transmission of a request and invoking of an operation in the appropriate *server* object.

Following the imperative paradigm, our approach uses synchronous communication if a server's method has to return a result. Otherwise the process of method invocation works asynchronously.

The two parts of an active object — *server* and *proxy* — are implemented as instances of the same class. The public member functions of that class make up the functionality of the proxy object. For each public function there is a corresponding private member function. The operations of the *server* object are implemented through those private functions. Furthermore, within a class of active objects at least one public and one private constructor exists. The public constructor is used to create the *proxy* object whereas the private one allows for creating the *server* portion of an active object. Calling the private constructor is done by an object of an additional class declared as *friend*. This way C++ rules of access control are obeyed. One of the public constructors which initiates initialization of the *proxy* object has a parameter of class string. If this constructor is called with the name of a node in the network as its argument, then it uses the *Object Space* to initiate creation of the *server* object on the specified node. This is handled by an instance of the *friend* class. The private constructor subsequently calls the *server* object's `scheduler` function. Afterwards, the *server* object may handle requests. The `scheduler` function does not return. It implements the server object's life.

Proxy and *server* objects exchange special protocol objects. These objects either carry an identifier for a private member function to be called in the *server* object together with the parameters needed to perform the call, or they carry the results of such a call. A corresponding class of protocol objects (derived from `Concurrency_prot`) has to be implemented for each class of active objects.

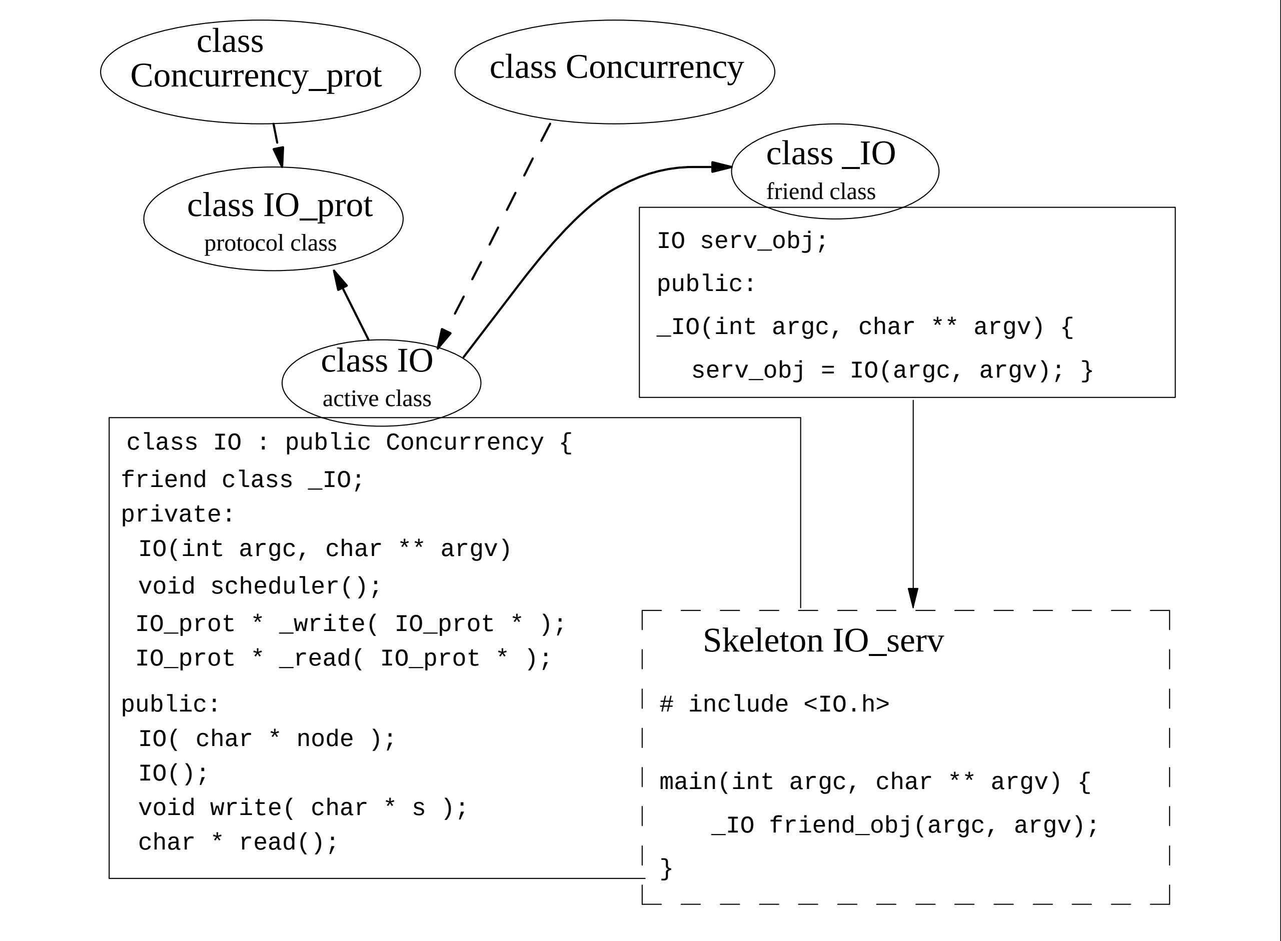
Calling a public member function in the *proxy* object results in storing a protocol object in the *Object Space*. The object has the description `request`. It denotes a private member function of the *server* object and contains the parameters needed by that function. Storing the protocol object in the *Object Space* is a non-blocking operation. So the *proxy* object may perform further steps if the server's operation does not return a result.

A synchronous member function of the *proxy* blocks after writing the protocol object with the operation request into *Object Space*. It awakes when the protocol object with the appropriate function results appears in the *Object Space*. The protocol object must contain the correct identifier for the *proxy* object and the description `answer`. After reading the protocol object its result value is returned to the caller of the *proxy*'s member function. Synchronous public member functions in a *proxy* object implement imperative communication between client and server. Additionally, member functions may work asynchronously. Asynchronous member functions are implemented using *future* objects. A non-void asynchronous member function returns a *future* object to its caller instead of the actual function result. Here we are using a concept similar to futures in ABCL/1 [Yonezawa et al. 87].

Future objects allow for delayed synchronization between the caller of one of the *proxy*'s member functions and the *server* object. Instead of a function result the caller receives a *future* object and may perform further operations. Within a *future* object there exists a typecast operation which allows assignment of the *future* object to a variable of the original result type. A class of *future* objects may exist for any possible result data type.

If a client tries to access the value of a *future* object, the typecast operation mentioned earlier is invoked. This operation may eventually block. In fact this typecast operation makes an attempt to read the appropriate protocol object from *Object Space*.

The strategy for dealing with requests within the *server* object is defined by function `scheduler`. The default version of this function is implemented in class `Concurrency`. It realizes a FIFO strategy. All the operation requests expressed as protocol objects of a particular class are stored in a common queue. Several *server* objects of a class share one queue of operation requests. So some kind of load balancing is implemented by the default `scheduler` function. Classes derived from `Concurrency` may implement their own versions of the `scheduler` function. Within such a specialized `scheduler` it is possible to associate *guards* with operations of the *server* object.



```
IO serv_obj;
public:
_IO(int argc, char ** argv) {
   serv_obj = IO(argc, argv); }
```

```
class IO : public Concurrency {
friend class _IO;
private:
 IO(int argc, char ** argv)
 void scheduler();
 IO_prot * _write( IO_prot * );
 IO_prot * _read( IO_prot * );
public:
 IO( char * node );
 IO();
 void write( char * s );
 char * read();
```

```
# include <IO.h>

main(int argc, char ** argv) {
    _IO friend_obj(argc, argv);
}
```

Fig. 5*: Instances of class IO are active objects and their proxies.*

Example: Distributed I/O system

We use an input/output system for distributed applications as an example to demonstrate how active objects work on top of *Object Space*. Let us consider a class `IO`. Instances of this class are active objects with their *proxies*. Within class `IO` there exist the public member functions `write()` and `read()`. They may be called via *proxy* objects. Their counterparts in the *server* are private member functions `_write()` and `_read()`. While `write()` works asynchronously, calls to `read()` involve synchronization between *proxy* and *server*.

Instances of class `IO_prot` serve as protocol objects. They handle the transmission of requests and results between *proxy* and *server* objects belonging to class `IO`. Finally the *friend* class `_IO` allows for initializing the *server* object. In addition to the three classes Fig. 6 shows the C++ code of the *Object Space* process belonging to the active object.

Class `IO` has one private and two public constructors. If a public constructor is called with the name of a node in the network a local *proxy* object is created. At the same time a new *Object Space* process is launched at the specified node. The process uses an instance of class `_IO` and the private constructor of class `IO` to initialize the *server* object. An additional parameterless constructor of class `IO` allows for the initialization of a *proxy* object without creating the corresponding *server* object.

Within Fig. 7 three *Object Space* processes are shown. These processes may run on different nodes within a network.

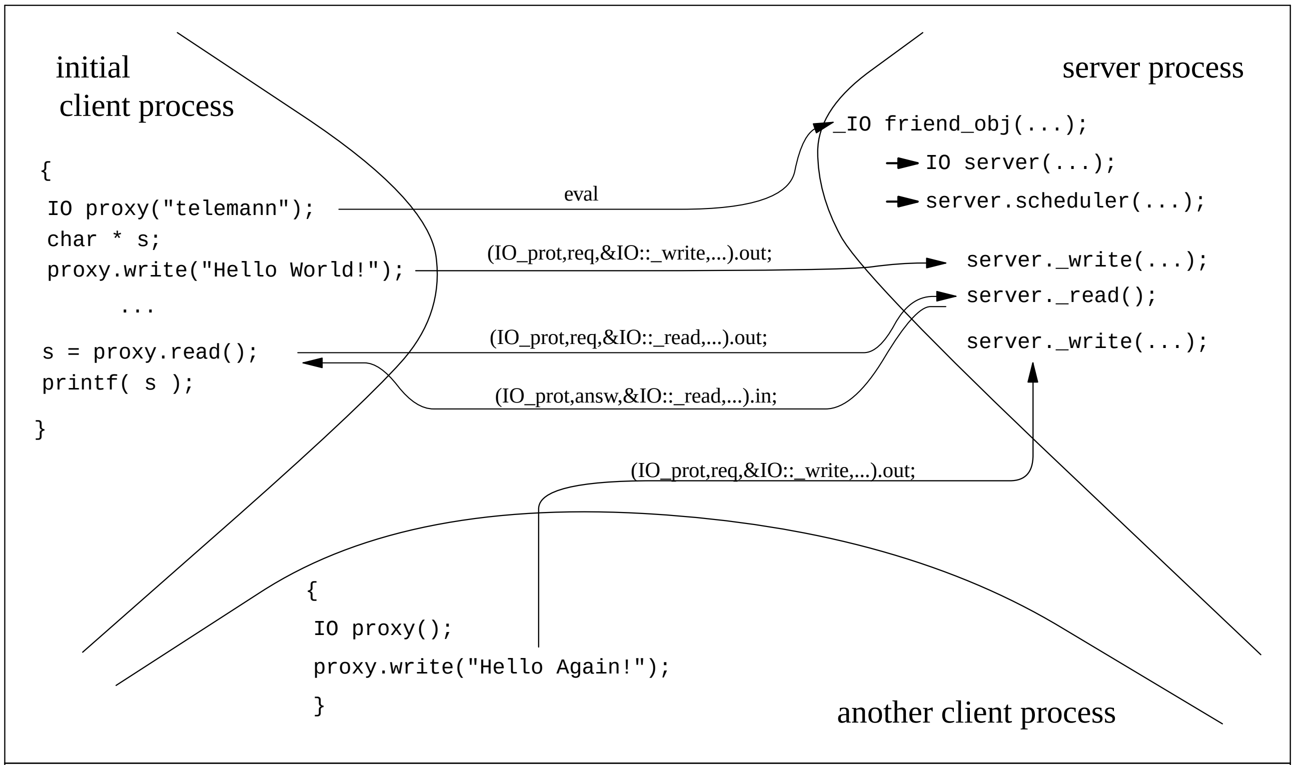


Fig. 7*: Several clients share an active object.*

While one of them implements a *server* object of class `IO`, both of the other processes contain *proxy* objects. One of the *proxies* has been constructed with the network node `telemann` as argument. Subsequently the *server* process has been created on node `telemann` using the *Object Space* operation *eval*. The second *proxy* object has been initialized by the parameterless constructor of class `IO`. As no explicit binding between sender and receiver of messages (objects) is needed in *Object Space* each *proxy* object may communicate with every existing *server* object of its class. Several clients may share the same *server* object via different *proxies*.

Furthermore, Fig. 7 shows the transmission of protocol objects between different components of a distributed application. The activation of private member functions in the *server* can be seen. When calling the asynchronous function `write()` in a *proxy* object, a protocol object denoting the operation request is written into *Object Space*. However, in `read`, which is synchronous, an attempt is made to read a protocol object of class `IO_prot` right after the operation request has been written. Therefore, `read()` immediately blocks until the *server* responds by writing the result into the *Object Space*.

If several *server* objects of one class exist, then it is not defined which of them handles a particular operation request. All these *server* objects share a common queue of operation requests. Therefore a new request is handled by the next available *server*. This can be regarded as some form of load balancing between several *server* objects.

If a sequence of operation requests has to be handled by the same *server* object, then a slightly modified communication protocol is needed. As the first step the *proxy* object has to obtain a key from the *server*. Then this key must be transmitted as part of the protocol objects. This establishes a virtual connection between *proxy* and *server*. Using the mechanisms of associative addressing within *Object Space*, the *server* object is able to receive only those operation requests which are initiated by the corresponding *proxy* with the correct key. So several operations may be handled continuously. After the virtual connection between *proxy* and *server* has been released (via another special operation), the *server* can now accept more operation requests from any *proxy* of its class.

5. Related Work

The *generative communication* style was first proposed within the context of *Linda* [Gelernter 85]. *Linda* was developed for programming parallel computers. It has, therefore, a slightly different focus than approaches intended for programming distributed systems. For example, costs for resources such as processes and communication delays are more important in distributed than in parallel environments. However, with respect to its decoupled communication style, the *Linda* model is well suited for distributed systems also. One drawback of *Linda* is that its communication constructs are rather low-level. This issue is addressed in [Ahmed 91]. Using object-oriented techniques to extend *Linda* with higher levels of abstraction has been the major idea behind *Object Space*.

Other approaches which combine *Linda*-like communication and object-orientation may be found in [Jellinghaus 90] and [Matsuoka 88]. In [Jellinghaus 90] the object-oriented language Eiffel is extended with Linda-like communication constructs. The result is called Eiffel-Linda. Eiffel-Linda supports usage of objects within an associatively addressed memory. Those objects are mapped onto tuples. The prototypical implementation of Eiffel-Linda relies on the original Linda runtime system from Yale. In Eiffel-Linda two objects may match only if they are instances of the same class. Inheritance is not taken into account. Corresponding non-void data components of two objects match if they have exactly the same values. Redefinition of the notion of ‘‘matching’’ is not supported within Eiffel-Linda.

In [Matsuoka 88] the integration of decoupled communication within the object-oriented language Concurrent Smalltalk is investigated. Associative addressing is implemented by calling special methods in the objects which take part in a tuple space operation. Thus an object may define how it matches against other objects. The addressing scheme described in [Matsuoka 88] does not take notice of inheritance between classes. In [Matsuoka 88] the tuple space is implemented as a special Concurrent Smalltalk object. Several instances of a tuple space may exist simultaneously. The prototypical implementation described in [Matsuoka 88] uses a single address space for the tuple space runtime system and all of its clients. A distributed version of the Concurrent Smalltalk tuple space is mentioned as a further research topic.

A discussion of interaction in distributed programs with several example algorithms may be found in [Andrews 91]. All examples presented rely on message-based communication, the communication style which presents the ‘‘lowest common denominator’’ among the styles discussed. In contrast, we have shown how message-passing may be transformed into *Object Space* communication by defining a `channel` class.

Several language designs exist which attempt to combine concurrency and object-orientation with an imperative communication style. We have discussed how this can be accomplished by introducing active objects to C++ through an external library on top of *Object Space*. A similar approach which starts with Eiffel and uses message-passing communication can be found in [Karaorman/Bruno 93]. Some examples of other approaches which either implement completely new concurrent object-oriented languages or extend existing languages by new constructs are Hybrid [Nierstrasz 87], μC++ [Buhr et al. 92], ABCL/1 [Yonezawa et al. 87], and CEiffel [Löhr 92].

6. Conclusions

Asynchronous message-passing and imperative client/server communication are two approaches for implementing communication between components of distributed applications. By defining C++ classes we have investigated how those communication styles may be transformed into decoupled communication as implemented by *Object Space*. The integration of object-oriented essentials and a decoupled communication style as realized in the *Object Space* approach prove to be well suited to the programming tasks commonly encountered in open distributed systems.

Message-passing with *send/receive* has been implemented on top of *Object Space* by defining a generic *channel* class. Typed channels may be represented by instances of classes derived from *channel*. An arbitrary number of processes may access a channel simultaneously. Since channels are represented by objects within *Object Space* they may be accessed from different nodes in a network in a transparent fashion. Thus, (re-)configuration of distributed applications using our channel class is simple. As an example we have discussed a distributed sorting program.

Active objects are a means to implement imperative communication in distributed client/server style applications. They combine concurrent and object-oriented programming. We have developed a class *Concurrency* which serves as the base class for active objects. Our implementation of active objects relies on *Object Space*, thus employing the decoupled communication style. This allows for a transparent sharing of active objects among several clients. Since no explicit binding between client and server objects takes place, a client does not need to know the server object's address nor is a special location brokering service needed. As an example we have described an I/O system for distributed applications based on active objects.

References

[Ahmed 91] S.Ahmed, D.Gelernter;
Program Builders as Alternatives to High-Level Languages;
Report YALE/DCS/RR-887, Yale University, Dept. of CS, November 1991.

[Andrews 91] G.R.Andrews;
Paradigms for Process Interaction in Distributed Programs;
ACM Computing Surveys, Vol 23, No.1, March 1991

[Buhr et al. 92] P.A.Buhr et al.;
μC++: Concurrency in the object-oriented language C++;
Software Practice and Experience 22, February 1992.

[Gelernter 85] D.Gelernter;
Generative communication in Linda;
ACM Transactions on Programming Languages and Systems, 7(1):80-112, 1985.

[Gelernter/Carriero 92] D.Gelernter, N.Carriero;
Coordination Languages and their Significance;
Communications of the ACM, Vol. 35, No.2, Feb. 1992.

[Hoare 74] C.A.R. Hoare;
Monitors: An operating systems structuring concept;
Communications of the ACM, Vol.17, No.8, August 1974.

[Karaorman/Bruno 93] M.Karaorman, J.Bruno;
Introducing Concurrency to a Sequential Language;
Communications of the ACM, Vol.36, No.9, September 1993.

[Löhr 92] K.-P.Löhr;
Concurrency Annotations Improve Reusability;
Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS) USA'92, Santa Barbara, August 1992.

[Matsuoka 88] S.Matsuoka, S.Kawai;
Using Tuple Space Communication in Distributed Object-Oriented Languages;
Proceedings of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA) '88.

[Nierstrasz 87] O.M.Nierstrasz;
Active objects in Hybrid;
ACM SIGPLAN Notices 22, December 1987.

[Polze 93a] A.Polze;
The Object Space Approach: Decoupled Communication in C++;
Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS) USA'93, Santa Barbara, August 1993.

[Polze 93b] A.Polze;
Using the Object Space: A Distributed Parallel make;
Proceedings of 4th IEEE Workshop on Future Trends of Distributed Computing Systems, Lisbon, September 1993.

[Jellinghaus 90] R.Jellinghaus;
Eiffel Linda: An Object-Oriented Linda Dialect;
ACM Sigplan Notices, Vol.25, No.3, December 1990.

[Yonezawa et al. 87] A.Yonezawa, E.Shibayama, T.Takada, Y.Honda;
Modelling and programming in an object-oriented concurrent language ABCL/1 in *Object-Oriented Concurrent Programming*;
M.Tokoro and A.Yonezawa, Editors, MIT Press, Cambridge, Massachussetts, 1987.