Objects, Replication and Decoupled Communication in Distributed Environments

Andreas Polze
Humboldt-Universität zu Berlin
Institut für Informatik
Lindenstraße 54a
10117 Berlin
apolze@informatik.hu-berlin.de

ABSTRACT

Decoupled communication has proven to be a powerful paradigm for the implementation of applications in the context of open distributed environments. We have developed the *Object Space* approach which integrates that communication style with object-oriented techniques. It allows encapsulation of protocols for interactions in distributed applications into classes, thus providing a new level of abstraction.

Besides an example application here we describe how decoupled communication as supported by the *Object Space* may be efficiently implemented in distributed environments. We briefly evaluate our prototypical implementation and develop a more efficient solution. This implementation is itself distributed. It runs on top of UNIX systems connected by a LAN.

1. Introduction

The *Object Space* approach to distributed computation allows for decoupled communication between program components by providing a shared data space of objects. The *Object Space* approach extends the sequential language C++ with coordination and communication primitives as known from *Linda*. It integrates inheritance into the associative addressing scheme and facilitates passing of arbitrary objects between program components. Thus classes may be used to describe a protocol for communication between components of a distributed application.

Object Space itself is implemented in a distributed fashion. It employs several *Object Space Manager* processes running on different nodes of a network under UNIX. We describe a naive prototype implementation of *Object Space*. Discussing how to avoid some performance drawbacks of this implementation, we develop an advanced solution. We use a technique called ‘‘optimistic asynchrony’’ to accelerate most *Object Space* operations.

In section 2 we briefly outline characteristics of the *Object Space* approach. A distributed application based on *Object Space* is shown in section 3. Section 4 presents a prototypical implementation of *Object Space*. In section 5 we discuss a more efficient distributed implementation of *Object Space* in greater detail and describe how our implementation may be adapted for use in the context of very large distributed systems. In section 6 we give an overview of related work and finally, in section 7 we present our conclusions.

2. *Object Space* Operations

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. *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*. This operation works asynchronously.
- **`in`** and **`rd`** carry a template (a special object) as argument. Both operations retrieve a matching object from the *Object Space* and store its values in the template; they work synchronously and may eventually block until a matching object is found. **`in`** removes the object from *Object Space*. An object matches a template depending on its class and 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 the address and arguments of a function which has to be executed locally as parameters.

Object Space relies on the abstractions ‘‘class’’ and ‘‘object’’. Objects serve as units of communication. C++ classes may be used for definition of communication protocols. These protocols describe the interactions between components of a distributed application. Previously defined protocols may be extended by use of inheritance. When dealing with objects in *Object Space* the C++ mechanisms of access control remain valid. Details of interaction may be hidden by construction of appropriate C++ classes. Those classes may provide secure interfaces to their clients. This view gives a new level of abstraction to the programmer of distributed applications.

The algorithm for associative addressing of objects within *Object Space* relies on an object's class and the values of its data components. A distributed type service is used to map classes' names onto unique integer identifiers. Inheritance may be expressed as a relation over those identifiers. Each pair ‘‘class name, identifier’’ is itself represented as a special object within *Object Space*.

3. Example: A distributed Time Service

We present a simple Client/Server application based on *Object Space* to show some advantages of object-based decoupled communication. In our example, time client and time server communicate by exchanging objects of a class `timprot`. Those objects are written into and retrieved from *Object Space*. They express a communication protocol — requests sent out by the client and answers sent by the server. Fig. 1 shows our example scenario.

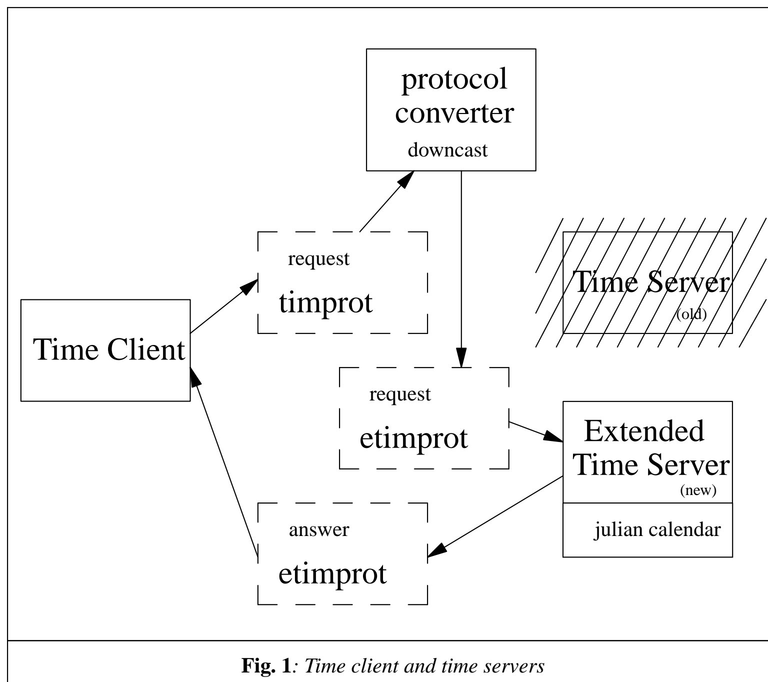

Fig. 1: *Time client and time servers*

However, the example in Fig. 1 is a little more then just client/server communication. We assume that during runtime our application has to be reconfigured. A new time server with extended functionality replaces the old one. This extended time server uses objects of class `etimprot` for communication with its clients. Class `etimprot` is derived from class `timprot` whose instances are expected to be used for communication by the time client.

The rules for associative addressing within *Object Space* allow the time client to read objects of class `etimprot` instead of `timprot`-objects. So the client can immediately receive answers from the extended time server. Unfortunately this does not hold for the opposite direction. To allow the time server to read objects of class `timprot` we have to perform a ‘‘downcast’’ . An object of a base class has to be converted into an instance of a derived class. To deal with this case an additional component is added to our distributed application — a protocol converter. It permanently attempts to get `timprot` objects from *Object Space* and transforms

them into corresponding objects of class `etimprot`. This way requests from the time client get transformed so that the extended time server can read and process them.

Two lessons can be learned from our example: First, no explicit binding between client and server is needed if both communicate via *Object Space*. So arbitrary many clients can communicate with any number of servers as far as all of them "speak" the same protocol (i.e. read and write objects of the same class). Load among several servers is balanced automatically. Secondly, distributed applications using *Object Space* can be re-configured even during runtime. In the distributed time service example we have exchanged the server component and the client did not even recognize it. The object-oriented characteristics of *Object Space* allow to extend previously defined communication protocols by inheritance.

4. A prototypical implementation

The *Object Space* approach supports the idea of a distributed shared memory. Our prototype implementation in C++ is based on interprocess communication mechanisms available within the UNIX operating system.

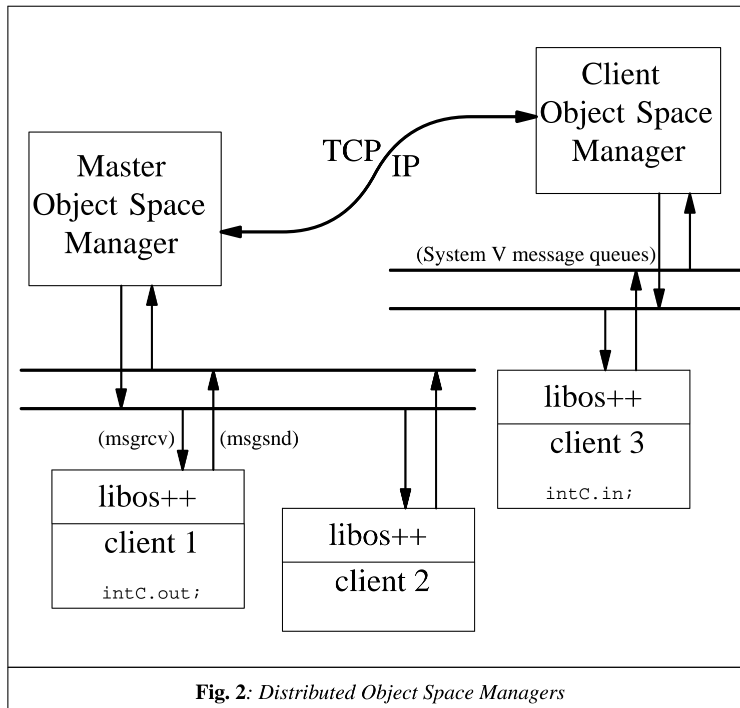

Fig. 2: *Distributed Object Space Managers*

A scenario around *Object Space* as shown in Fig. 2 includes several processes, each of them storing objects into and retrieving them from *Object Space*. Some kind of storage medium is needed to keep the data which makes up an

object available. In our implementation these storage media are provided by a separate UNIX process, the *Master Object Space Manager*. It uses virtual memory to store the objects. Furthermore this process performs matching between templates and objects. In addition to the *Master Object Space Manager* several *Client Object Space Manager* processes may exist. These processes do not store any object at all. They just forward operation requests to the *Master Object Space Manager* process using a special protocol on top of TCP/IP. Thus crossing machine boundaries is possible.

Client processes communicate with a local *Object Space Manager* process. This process may be either the *Master Object Space Manager* or a *Client Object Space Manager* as shown in Fig. 2 . They use message queues as the communication mechanism. Message queues have been introduced with UNIX System V but currently they are available with nearly every UNIX System. A library `libos++` transforms calls to *Object Space* operations into a proper sequence of `msgsnd()` and `msgrcv()` system calls on these message queues. Client programs may access the operations from `libos++` through inheritance from two communication base classes. An *Object Space Manager* handles two message queues, a read queue and a write queue. Each client process contacts the read queue of a local *Object Space Manager* to initiate operations on *Object Space*. Results of operations are returned on the corresponding write queue.

The operation **`out`** initiates a write-operation on the read-queue of a local *Object Space Manager*. It transmits an *actual* object and does not return a result. On the other hand, the operations **`rd`** and **`in`**, called on client side, write a template object onto the read-queue of a local *Object Space Manager* and block until it returns a matching object. The matching object is returned on the corresponding write-queue. Both read- and write-queue are multiplexed, thus allowing any number of clients to communicate with the same *Object Space Manager* at once.

Matching between templates and *actual* objects is performed by the *Master Object Space Manager*, so this process is a potential performance bottleneck. But we discover another performance problem when looking at communication delays. Practical experience has shown that communication via message queues is much faster than via a TCP/IP connection. We have measured that one *Object Space Operation* takes roughly 1.05 ms on a DECstation 5000/125 if communication occurs over message queues only. It takes about 10 ms if local TCP/IP communication (which is entirely handled in software) is involved. And finally, the same *Object Space Operation* issued on a remote site, performed using message queues and TCP/IP, takes about 21 ms. We have obtained similar results on Sparc 2. On a 386based PC-Unix the operations took about twice the time. So our prototype implementation of *Object Space* is acceptable on multiprocessor systems where all communication can be performed via message queues. For a real distributed environment our prototypical implementation seems to be suitable only if *Object Space* operations are performed infrequently.

5. A more efficient implementation of *Object Space*

The *Master Object Space Manager* plays an integral role within the prototypical implementation of *Object Space* as described in section 3. It maintains all the objects contained in *Object Space* and performs the matching between templates and actual objects. As each call to an *Object Space* operation has to pass this process, it may soon become a performance bottleneck. To deal with this problem we replicate data. Objects are stored within several *Object Space Manager* processes. So each of the synchronous operations **`in`** and **`rd`** has to access local data only. Fast communication via message queues is used in that case. Another problem is ensuring mutual exclusion. Although two copies of a certain object may exist in different *Object Space Managers*, only one client performing an **`in`** operation must be able to obtain this object. To achieve this we need a special protocol for communication between *Object Space Managers*.

In general, replication of objects in several *Object Space Managers* is useful only if those objects are accessed from different nodes in the network at nearly the same time. To implement associative addressing in the *Object Space*, we have employed a distributed type service. Prior to each *Object Space* operation concerning an instance of class ‘‘x’’, the type identifier for that class has to be obtained. At least the first creation of an instance of class ‘‘x’’ in a program component includes communication between the particular program component and an *Object Space Manager*. Thus, an *Object Space Manager* is able to know the classes of objects which are used by its local clients (components of a distributed application). Employing this knowledge, a rule for replication of objects on different *Object Space Managers* may be defined. This rule has to consider the associative addressing scheme and inheritance. It consists of two parts:

1) ‘‘A copy of each object of class ‘x’ contained within *Object Space* has to be stored on a node if at least one *Object Space* operation using an instance of class ‘x’ as argument was performed on this node.’’

2) ‘‘A copy of each object of class ‘y’ contained within *Object Space* has to be stored on a node if at least one operation using an instance of a class ‘x’ which is ultimative base class of ‘y’ was performed on this node.’’

When formulating this rule we assume that a node is represented by an *Object Space Manager* process. Furthermore we assume that *Object Space Manager* processes on different nodes are fully interconnected.

In Fig. 3 four processes on different sites called Node A trough Node D are fully connected via TCP/IP communication links. Those processes implement the *Object Space*. They store instances of classes `a`, `b`, `c` and `d`. Class `d` is derived from class `c`. In Fig. 3 a set of classes is associated with each node. These sets describe classes whose instances are stored on a node according to the replication rule formulated earlier. The *Object Space* operations printed within the node’s boxes in Fig. 3 are performed by clients of the corresponding *Object*

Space Managers. For simplicity we have omitted the clients of *Object Space* from our drawing. Instead of declaring variables of the appropriate classes we have used the classes' names to indicate *Object Space* operations here.

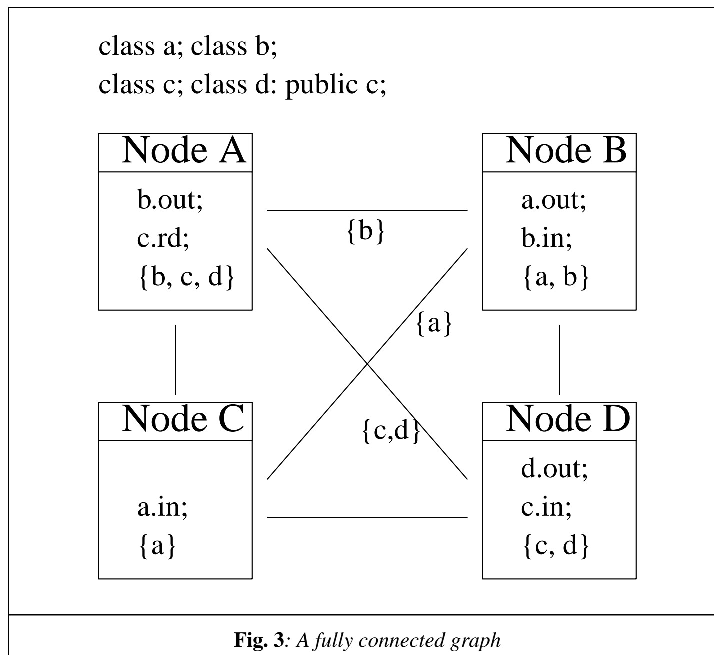

Fig. 3*: A fully connected graph*

One may notice that Node A stores instances of class `d`, although no *Object Space* operation concerning this class has happened on Node A. But due to the associative addressing scheme used within *Object Space* the operation `c.rd` may eventually return an object of class `d`. Communication links between *Object Space Manager* processes are used to run a protocol to ensure consistency among replicated objects. In Fig. 3 the links are labeled showing the classes whose instances are transmitted via a particular link.

Now we want to discuss in greater detail how the *Object Space* operations **`out`** and **`in`** work in context of our replication scheme. An unique object identifier and a replication counter are assigned to each object and its replicas within *Object Space*. The node where an object initially appeared (via operation **`out`**) is called an object's host. The unique identifier for an object and its replicas is composed from the host name and a serial number. Between *Object Space Manager*s, our algorithm uses message passing via TCP/IP to distribute an object's copies. Operations **`out`** and **`in`** employ message passing between remote *Object Space Managers* whereas operation **`rd`** works locally. Four different types of messages are used.

Replication of objects may be achieved by transmitting the object's data to all *Object Space Manager*s which store instances of the object's class. Thus

Object Space operation **`out`** results in broadcasting of messages of type ''out''. These messages contain the object's identifier and its data.

The implementation of operation **`in`** has to ensure mutual exclusion among several *Object Space Managers* attempting to access the same object. As mentioned above for each object with its replicas a particular *Object Space Manager* serves as the object's host. That object's host has to resolve conflicts resulting from concurrent **`in`**-operations for the same object. Before actually performing an operation **`in`** on a particular object an *Object Space Manager* has to send a request to the object's host. Those requests are handled on a first-come-first-serve basis.

A message of type ''initiate-in'' sent to an object's host expresses the request of a remote *Object Space Manager* to get exclusive access to that object. The object's host may answer with a ''commit-in'' message. Then the remote *Object Space Manager* may safely forward the object to its client, the operation **`in`** is completed. Additionally ''delete-copy'' messages are sent by the object's host to all other holders of replicas of the object.

Thus, an *Object Space Manager* which issued an ''initiate-in'' message for a particular object may receive either a ''commit-in'' message or a ''delete-copy'' message regarding that object. Then it can either forward the object to its client or it has to find another matching local object and to perform the above steps again.

The operation **`in`** implemented as described above requires at most exchange of two IP-messages between a remote *Object Space Manager* and an object's host. We can now attempt to figure out costs (i.e. communication delays) of *Object Space* operations.

- From a client's view, operation **`out`** may be handled asynchronously. A client simply stores an object on its local *Object Space Manager* and may continue execution. This operation involves message queue communication only and therefore has very little delay. The *Object Space Manager* distributes the new object via the communication links labeled with the object's class name and attaches object identifier and replication counter with the object.
- The operation **`rd`** is a synchronous operation. Due to our replication scheme this operation may act on data stored in the local *Object Space Manager*. Thus only fast message queue communication needs to be used. Because operation **`rd`** has no impact on the contents of *Object Space*, no communication between *Object Space Manager* processes is needed to perform that operation. Operation **`rd`** may eventually block. In that case the unsatisfied operation request is stored in the local *Object Space Manager*.

- The operation **`in`** also works synchronously. A client process has to wait until the local *Object Space Manager* returns a matching object. Before returning an object, the *Object Space Manager* has to request that particular object from its host. But unfortunately, slow TCP/IP communication has to be used by the *Object Space Manager*s to negotiate about a particular object. In the case of a blocking **`in`** operation, i.e. the local *Object Space Manager* has no matching object available, the unsatisfied operation request is stored.

With our new implementation scheme we have shown, how *Object Space* operations **`out`** and **`rd`** may be realized with little delay using fast message queue communication and locally stored replicas of objects. Now we want to discuss how the performance of operation **`in`** can be further improved.

When performing an **`in`** operation an *Object Space Manager* chooses a matching object, forwards it to the client and deletes it from the *Object Space*. Since the object may be replicated, the object's host has to commit to the above steps. All additional holders of replicas have to delete their copies of the object, too. Our replication scheme guarantees that as few replicas as possible are stored within the system. However, the communication between an *Object Space Manager* and an object's host increases the time an **`in`** operation takes.

Instead of waiting for commitment from a particular object's host, we may forward the object to the client process early. Then the client process may continue execution. Eventually this may be wrong since the *Object Space Manager* may fail when negotiating with the object's host. So the client process must be able to set back to the point of execution where it received the object from *Object Space*. Then it must be able to continue execution with another object. A transaction-like mechanism is needed to implement this behaviour. Our approach softens the semantics of operation **`in`**. We call this technique "optimistic asynchrony" since the result of an otherwise synchronous operation is used before the operation is entirely completed.

To ensure consistency within *Object Space* a client returning from an **`in`** operation gets blocked when performing the next *Object Space* operation until all *Object Space Managers* have committed about deletion of the **`in`** operation's object from *Object Space*. Nevertheless we are able to perform **`in`** operations with the delay of message queue communication only if a client does not perform sequences of *Object Space* operations, an assumption which is true in many cases.

Setting breakpoints during the execution of a client's process and returning to such a breakpoint can be implemented by different means. The UNIX system provides C library routines `setjmp/longjmp` which allow to restore the stack of a process. An extra effort has to be made to keep a process' heap consistent when performing a call to `longjmp`. In [Bilris at al. 93] the authors discuss how overloading of operator `new` in a C++ class may be used

to keep track of a process' heap. The technique described there is used to implement persistent objects in a database system. Overloading of operator `new` for all classes whose instances appear as arguments to *Object Space* operations allows to take care of the heap management when calling `longjmp`. To implement a transaction scheme we additionally need an inverse operation for each function which uses the result of an **`in`**-operation. From a client's point of view, once these operations have been specified, setting a process back to a breakpoint may happen fully transparently.

Another way to implement "optimistic asynchrony" would be to keep an invisible reference to the object returned from operation **`in`** in the library `libos++`. Later on, if negotiation about the object retrieved from *Object Space* has failed, this reference may be used to modify the previously retrieved object. The client process has to support a notification function which is called by the library `libos++` in such a case. This technique does not work transparently, the programmer of a client process has to deal with a "failed" operation **`in`** explicitly. However, the notification function is somewhat similar to an inverse operation in the transaction scheme mentioned earlier. But it allows greater flexibility. For example, instead of un-doing all the operations on a particular object the programmer can decide to simply delete the object. To avoid the extra effort introduced by dealing with a notification function, a client process' programmer may enforce operation **`in`** to work synchronously. Thus the original semantics of that operation may be obtained.

We have presented an implementation scheme for the *Object Space* approach which allows performance of *Object Space* operations with basically the costs of local interprocess communication even in distributed environments. To achieve this result we employ replication of objects stored in *Object Space* and "optimistic asynchrony" of **`in`** operations. Our replication scheme guarantees as few replicas as possible of a particular object within the system. Additionally, with our new implementation scheme we are able to address another topic: fault tolerance.

Fault tolerance

In open distributed environments components of a system may disconnect from the system — services can be unavailable for some amount of time. The *Object Space* approach should be able to deal with those cases. Although the breakdown of a network connection may be due to a scheduled downtime and thus be intended, we consider it as a fault. Our implementation of *Object Space* is able to respond to certain faults through reconfiguration.

We consider two kinds of failures: the breakdown of a network connection and the breakdown of a node, represented by an *Object Space Manager* process. Since we use TCP/IP as the communication mechanism, both cases can be easily detected by a surviving *Object Space Manager*s. In those cases the UNIX system call `read()` returns a value of 0. When receiving such a

result a process tries to establish a connection to the initial communication partner via a third node. This succeeds if the initial partner survived and results in a reconfiguration as shown in Fig. 4 .

In our example shown in Fig. 4 the direct connection between Node B and Node C is assumed to be faulty. All communication regarding operations **`out`** and **`in`** for objects of class a now has to be performed via Node D. This node acts as a bridge. The functionality of *Object Space* is not affected by that reconfiguration.

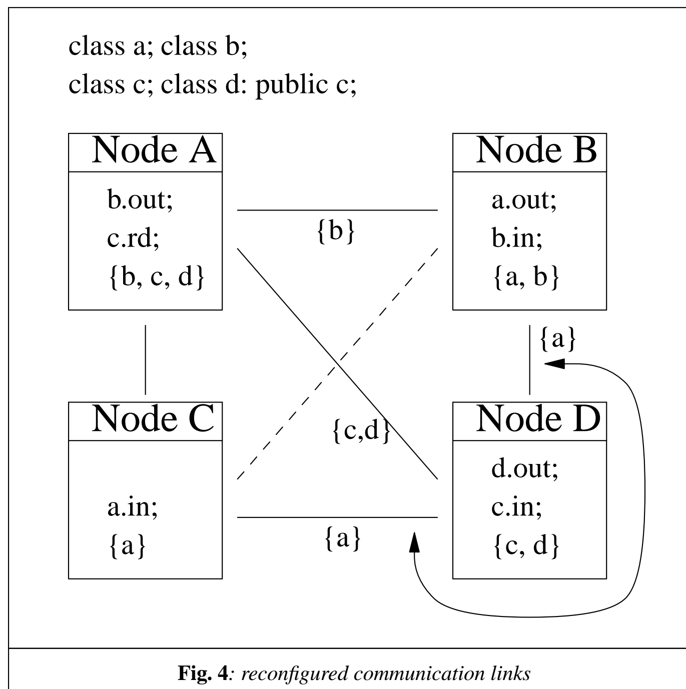

Fig. 4*: reconfigured communication links*

In general, if a node cannot reach a particular communication partner it subsequently asks its remaining partners for an connection to that particular node. If none of them succeeds the node is considered to be faulty. In our example, if Node C cannot reach Node B, neither via Node D nor via Node A, then Node B is considered to be faulty (or disconnected). A new host has to be assigned to objects whose host disappeared from an *Object Space* segment. Our fully connected graph with four nodes from Fig. 4 degenerates in that case into a fully connected graph with three nodes and a single node. Clients of *Object Space* using only Node A, Node C and Node D can proceed as usual. On the other hand, if we assume that Node B is simply disconnected for some reason, clients of Node B can proceed too.

An *Object Space Manager* periodically attempts to reconnect to each of its former communication partners. Within our example, the connection from Node B to another node may become re-established. Then this node

would act as a bridge. Objects with the same identifier stored on Node B and on any other node would be considered to be replicas. Then only the object's host attached to the replicas of such an object would have to be adjusted. Objects of class b which are stored on Node B but not on Node A would be replicated onto Node A and vice versa.

We have shown, how our new implementation scheme of *Object Space* can deal with network failures. Furthermore, we have described a method whereby the temporal disconnection of a node may be handled by the *Object Space*. Both cases are common in open distributed systems. Thus the predicate ''fault tolerant'' describes the usability of our approach in the context of open distributed environments.

Large Scale Distributed Systems

Our new implementation scheme for the *Object Space* employs replication of objects and optimistic asynchrony to minimize delays when accessing objects stored in *Object Space*. This approach is well suited for small scale distributed systems. Operations **`out`** and **`rd`** are delayed by local interprocess communication only. Execution of operation **`in`** requires exchange of two TCP/IP messages.

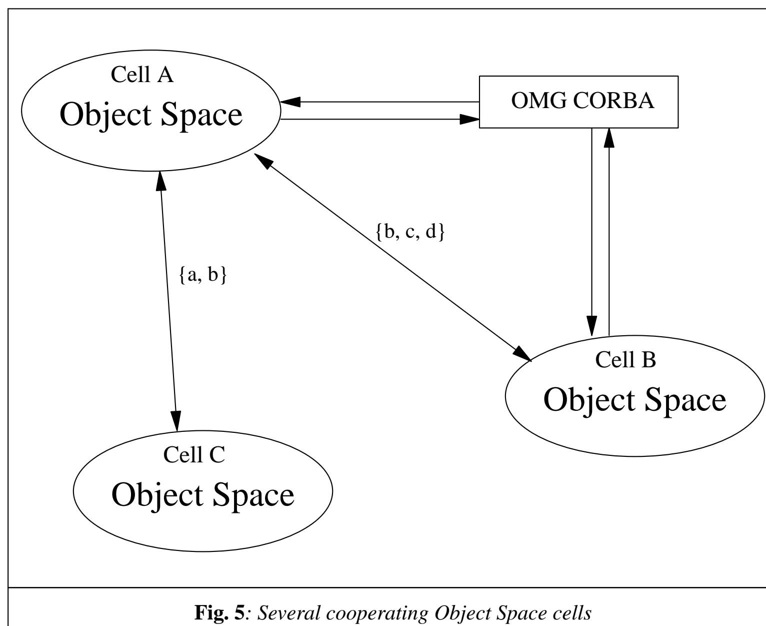

Fig. 5*: Several cooperating Object Space cells*

In large scale distributed systems cost of communication (delays) should not be hidden from the programmer. So we suggest to subdivide a *Global Object Space* for large scale distributed systems into a set of cells. Each cell can be implemented as an *Object Space* as described above.

Communication within a cell is relatively cheap — it can be performed with small delays.

Whereas our original implementation of *Object Space* assumes a static interconnection between *Object Space Manager* processes, connections between cells are considered to be dynamic. We want to employ some kind of "request brokering" (e.g. OMG CORBA) to allow clients of *Object Space* to build up connections between cells.

Communication between cells is more expensive than within a cell. So we distuingish between heavy-weight objects, which are distributed among cells and light-weight objects, which reside inside a single cell only. Whereas light-weight objects can be considered as memory elements in a shared memory, heavy-weight objects are comparable to RPC-invocations — mainly thought to express requests and results of remote services. The programmer can control wether objects are locally or globally distributed by simply deriving them from different communication base classes.

6. Related Work

The *generative communication* style was first proposed within the context of *Linda* [Gelernter 85]. *Linda* was developed for programming parallel computers and therefore has a slightly different focus than approaches intended for programming distributed systems. However, as well as a parallel implementation in [Bjornson et al. 1987] a distributed implementation of *Linda* is also described. This implementation makes use of various preprocessing techniques embedded within the *Linda* precompiler. The precompiler analyzes a parallel program with all its components as a whole, a technique which is not usable in the context of open distributed systems. A further description of precompiler techniques may be found in [Carriero/ Gelernter 91].

One drawback of *Linda* is that its communication constructs are rather low-level. This issue is addressed in [Ahmed 91]. The use of 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]. However, in these papers no issues regarding a real distributed implementation are mentioned.

In [Callsen et al. 91] two distributed implementations of a *Linda*-like system are described. One of them runs under the UNIX operating system, the other under Helios on top of transputers. The transputer ver-

sion uses a ring topology for communication between nodes storing tuples. Within that system, requests for tuple space operations travel along the ring from node to node, eventually matching against tuples already stored on a particular node.

7. Conclusions

We have described two distributed implementations of *Object Space*. The first, a naive prototypical implementation, shows some performance drawbacks due to a centralized *Master Object Space Manager* process and communication delays caused by TCP/IP. However, this implementation seems to be suitable for multiprocessor systems running UNIX where all interprocess communication can be handled via message queues.

In the second, more efficient implementation *Object Space* is realized as a fully connected graph of *Object Space Manager* processes. This implementation introduces a replication scheme for objects. Together with "optimistic asynchrony" of **`in`** operations, this technique allows most *Object Space* operations to be performed without delays caused by communication with a remote site.

In the context of our new efficient implementation we have discussed how network failures and breakdown of nodes in a distributed environment may be handled. We have proposed an algorithm whereby our new implementation of *Object Space* may deal with those cases. Furthermore, we have briefly discussed how our implementation of *Object Space* may be adapted for very large distributed systems.

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.

[Bilris at al. 93] A.Bilris, S.Dar, N.H.Gehani; *Making C++ Objects Persistent: the Hidden Pointers*; Software-Practice and Experience, Vol. 23(12), pp. 1285-1303, December 1993.

[Bjornson et al. 1987] R.Bjornson, N.Carriero, D.Gelernter, J.Leichter; *Linda, the Portable Parallel*; Research Report YALE/DCS/RR-520, February 1987.

[Callsen et al. 91] C.J.Callsen, I.Cheng, P.L.Hagen; *The AUC C++ Linda System*; in Technical Report 91-13, Edinburgh Parallel Computing Centre, pp. 39-71, Greg Wilson (Editor).

[Carriero/Gelernter 91] N.Carriero, D.Gelernter; *New Optimization Strategies for the Linda Pre-Compiler*; in Technical Report 91-13, Edinburgh Parallel Computing Centre, pp. 74-82, Greg Wilson (Editor).

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

[Jellinghaus 90] R.Jellinghaus; *Eiffel Linda: An Object-Oriented Linda Dialect*; ACM Sigplan Notices, Vol.25, No.3, pp. 70-84, December 1990.

[Matsuoka/Kawai 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, pp. 276-284.

[Polze 93a] A.Polze; *The Object Space Approach: Decoupled Communication in C++*; Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS) USA'93, pp. 195-204, 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, pp. 234-239, Lisbon, September 1993.

[Polze 94a] A.Polze; *Interactions in Distributed Programs based on Decoupled Communications* in Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS) USA'94, pp. 81-91, Santa Barbara, August 1994.

[Polze 94b] A.Polze;
Objektorientierung und lose gekoppelte Kommunikation als Basis für die Entwicklung offener, verteilter Anwendungssysteme; Dissertation am Fachbereich Mathematik und Informatik der Freien Universität Berlin, verteidigt am 10. Oktober 1994.