Objects, Replication and Decoupled Communication in Distributed Environments

Andreas Polze
apolze@informatik.hu-berlin.de

Humboldt-Universität zu Berlin
Institut für Informatik
10099 Berlin

Overview

- The *Object Space* Approach:
  object-oriented essentials and decoupled communication.
- Example: a distributed time service.
- A distributed protoypical implementation of
  Object Space.
- Employing replication and optimistic asynchrony:
  a more efficient implementation.
- Parallel Computing in Distributed Environments:
  Shared Objects Memory

The *Object Space* Approach

- Distributed shared memory serves as object store.
- Decoupled communication style — inspired by *Linda* + tuple space.
- Integration of object-oriented essentials like inheritance and data encapsulation.

  → (re-) configuration of an application is simple.

  → Objects as units of communication.

  → (UNIX) processes as units of distribution.

  → Associative addressing of objects.

    — No global name space needed, robustness, openess.

  → Access to an object does not depend on the node a process is running on.

  → Objects may be manipulated within *Object Space* as a whole only.

The *Object Space* Language

```
OSL-Program    ::={ object_space_op | local_computation }.
object_space_op::=object "." op_spec.
op_spec        ::=rd | in | out | eval .
object         ::=[ objID ]"(" class [ ":" base ] "," data_comp
                   { "," data_comp } ")".

data_comp      ::=formal | actual.
formal         ::=[ type ] "?" name.
actual         ::=[ type ] name [ "=" value [ matching_fct ]].
matching_fct   ::= delta"(" diff ")"
                   | or value { or value }.
```

- *Coordination language* — embedded into C++.
- Associative addressing considers inheritance.
- Introduction of *matching* functions.

> Let be a template's data component,
> let for 1 ≤ i ≤ n be alternative values of a template's data component,
> let be a component of an object stored within *Object Space*:
>
> **`or`**

- Four operations:

  `out` stores an object into *Object Space*.

  `rd` reads an object.

  `in` reads an object and deletes it from *Object Space* in an atomic operation.

  `eval` creates a new process, either locally or remotely.

- Data encapsulation, inheritance.

  → Usage of proper abstractions.

- Runtime type service

  — Classes identified by their names.

  — Distributed type service maps names onto integer IDs.

  — Inheritance is expressed as relation over those IDs.

- Higher level of abstraction: service, protocol — in open environments.

  → Advantage over approaches as *Linda Program Builder*.

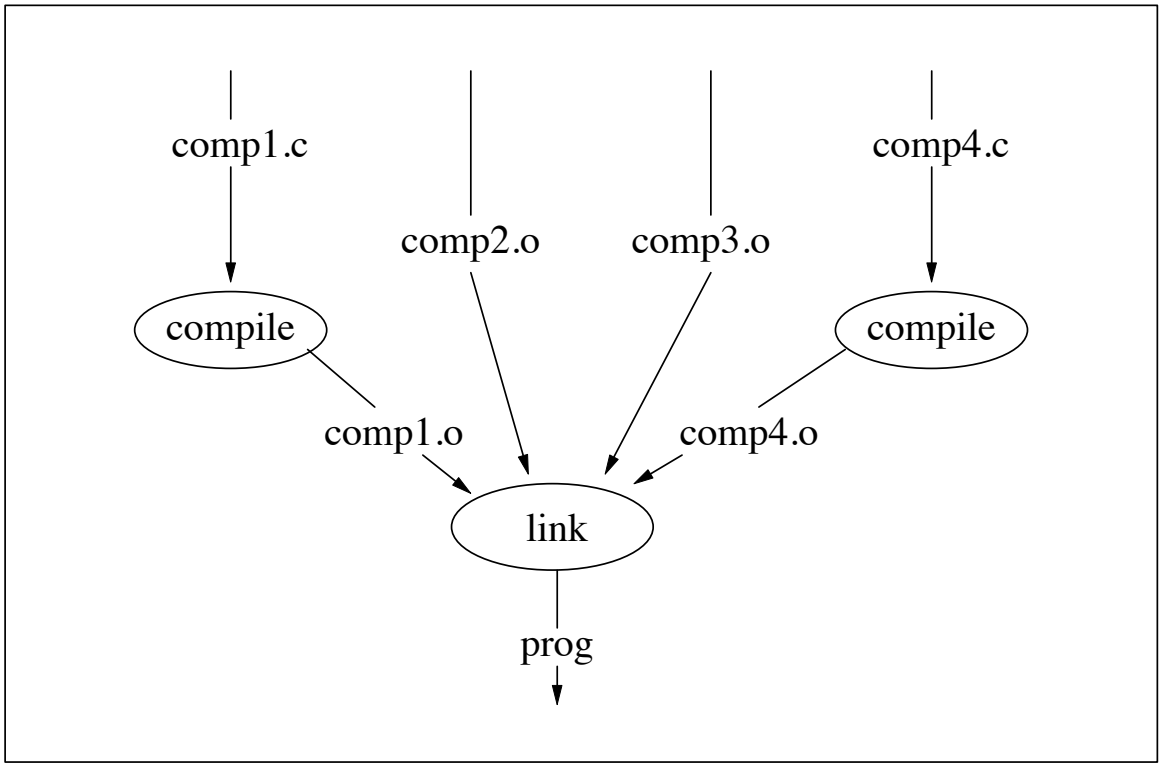
Example for *OSL*

```
(tstatC, f="comp1.o", s="nonexistent").out;      (1)
(tstatC, f="comp2.o", s="up-to-date").out;
(tstatC, f="comp3.o", s="out-of-date").out;
tstatC o;                                        (2)
o(tstatC, ?f, s="nonexistent"
            or "out-of-date").rd;                (3)
o.show_f();                                      (4)
```

- *OSL* constructs are transformed into C++.
- Calls to functions `actual()` and `formal()` deliver structure information about objects.

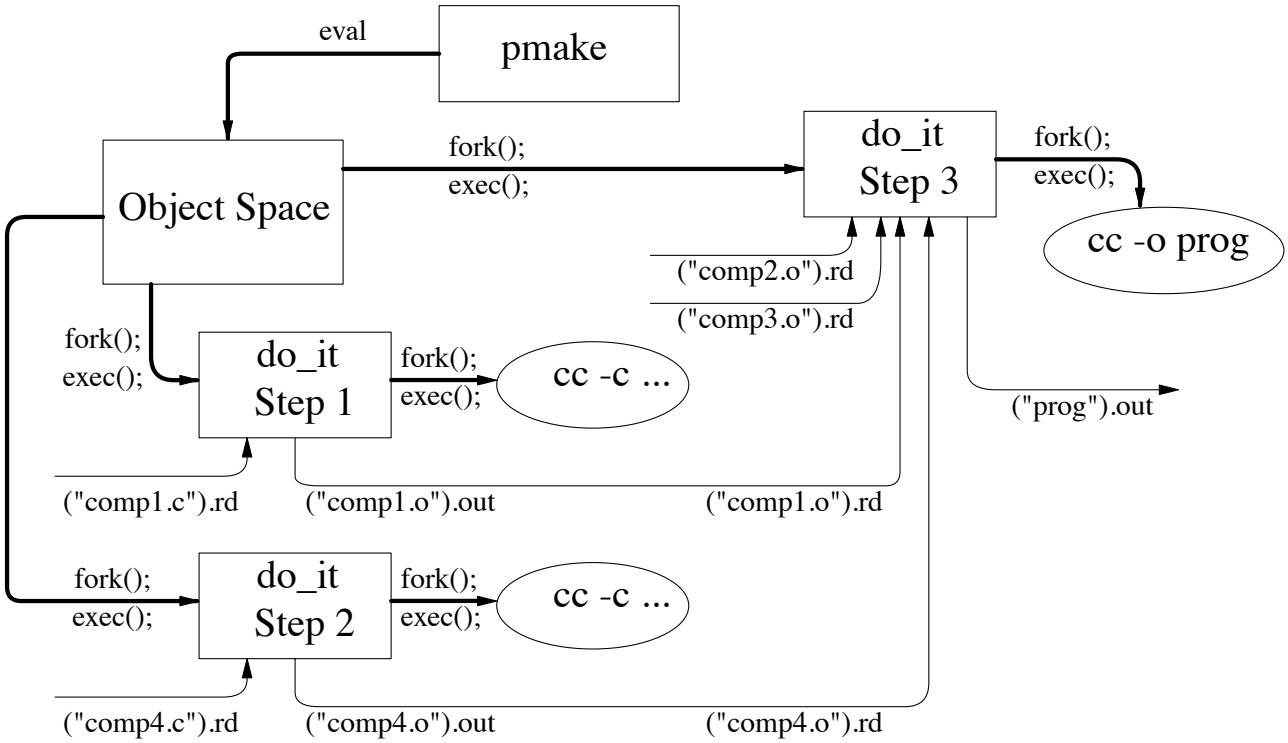
Example: a Parallel Distributed `make`

- Actions are modeled within a directed acyclic graph.
  - — nodes represent actions.
  - — edges represent data dependencies.
- Instances of class `tstatC` describe state of sub-targets and define communication protocol.



- For each input dependency which is up-to-date an object is stored in *Object Space*.

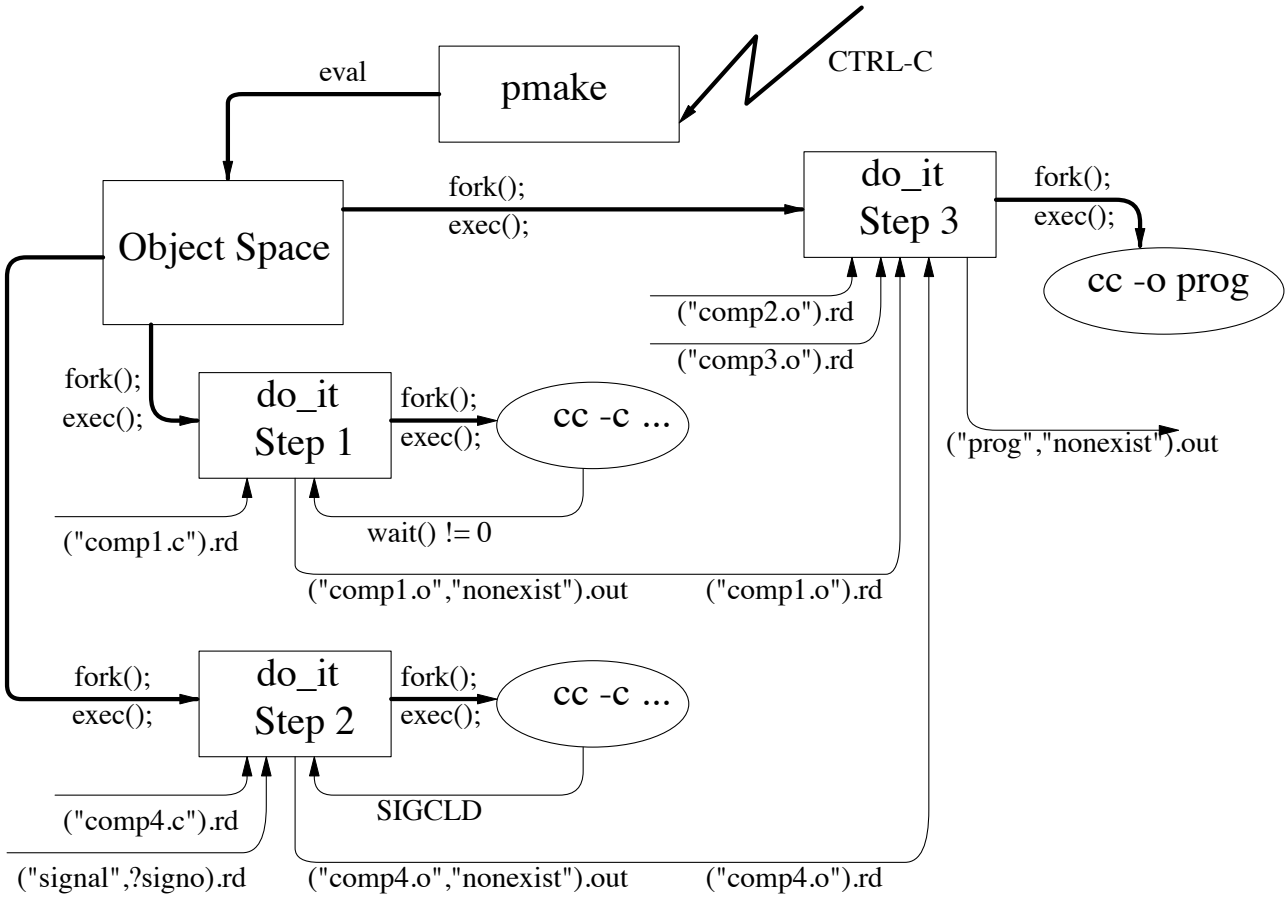
```
(tstatC, f="comp1.c", s="up-to-date")
(tstatC, f="comp2.o", s="up-to-date")
(tstatC, f="comp3.o", s="up-to-date")
(tstatC, f="comp4.c", s="up-to-date")
```



- Abnormal termination of make actions has to be handled.
- Several applications using one instance of *Object Space* have to cooperate — unique names.

Dealing with abnormal termination

- Compiler or linker steps may fail.
- User may interrupt the `make` program (CTRL-C).



→ Distributed termination is difficult.

- Non-blocking predicates **`rdp()`** and **`inp()`**.
- Delivering of signals to remote processes via *Object Space*.

Distributed Time Service

- Client/Server scenario.
- Communication protocol has to be defined.
  - → Introduction of class `timprot`.
  - → `timprot` objects express requests and answers.
  - → UNIX processID as identifier for a certain client used.
- Objects may be manipulated as a whole only.
- Concept of member functions allows construction of secure interfaces.
- Aspects of distribution may be hidden from application programmers.
- Open environments supported: sender of a request has not to know its receiver nor vice versa.

```
enum { REQUEST, ANSWER } operT;

class timprot: public objsp_comm {
protected:
  char * tstr;
  operT  op;
  int    magic;

  timprot():objsp_comm("timprot") {
    timestr=NULL; op=REQUEST; magic=getpid();
  }
};
```

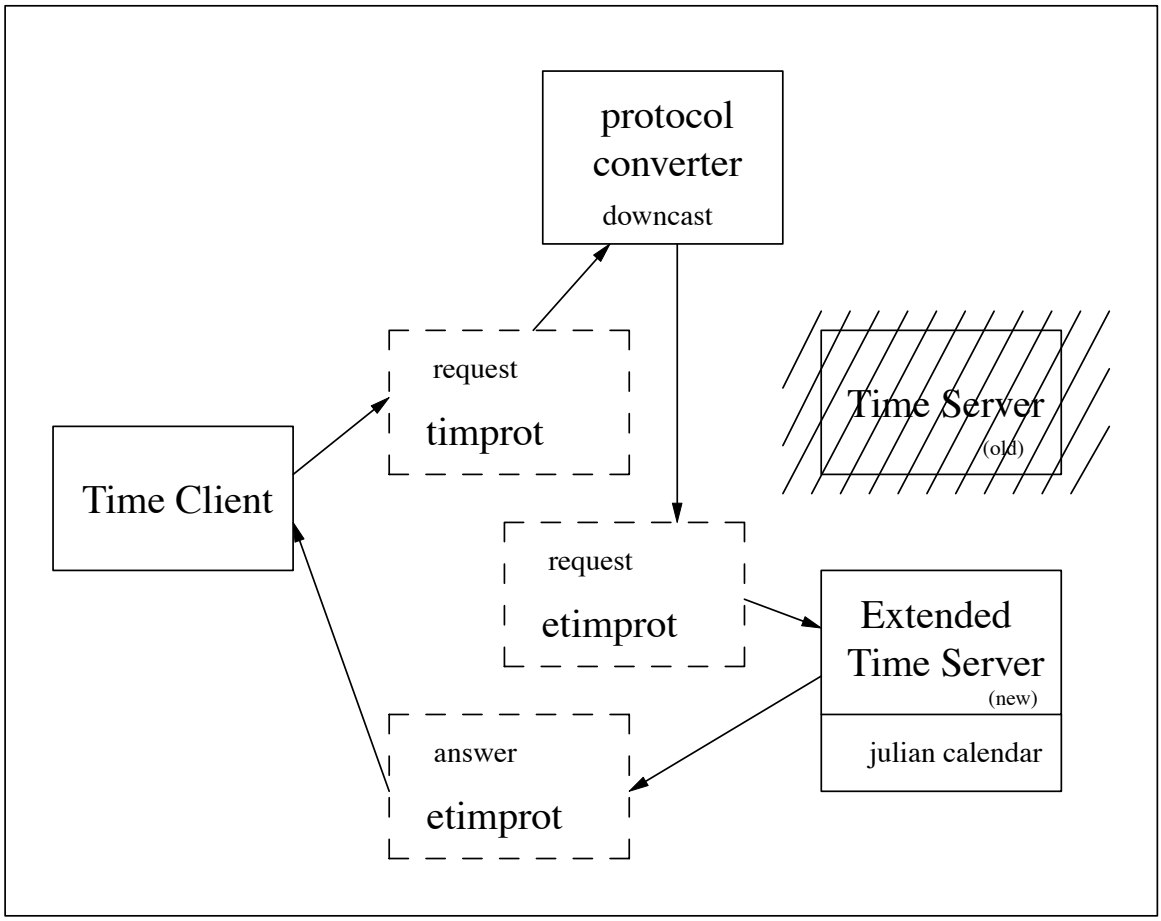
```
class timc: public timprot {
  void ask_server() {
    timprot tim_obj;
    tim_obj(timprot,tstr,int op=REQUEST,magic).out;
    tim_obj(timprot,?tstr,int op=ANSWER,magic).in;
  }
public:
  timc() { ask_server();}
  char* show() { return tstr;}
};
```

```
main() {
  timc client_obj;

  printf("the time is %s\n", client_obj.show());
}
```

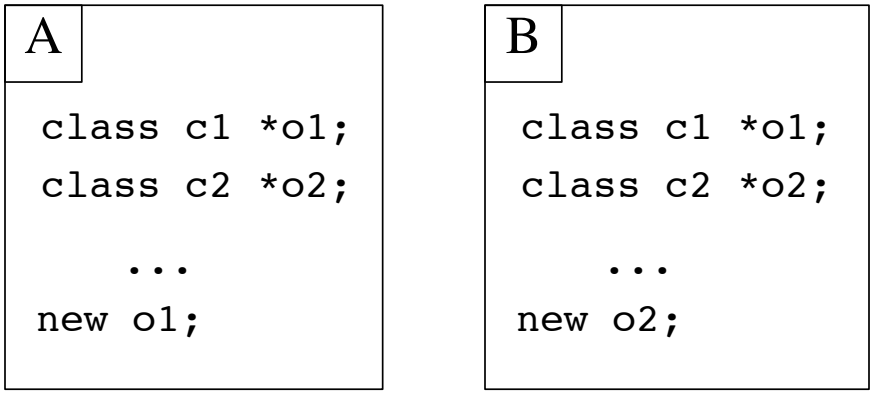
Extension of a communication protocol

- Time server with extended functionality.
- Class `etimprot` is derived from class `timprot`.



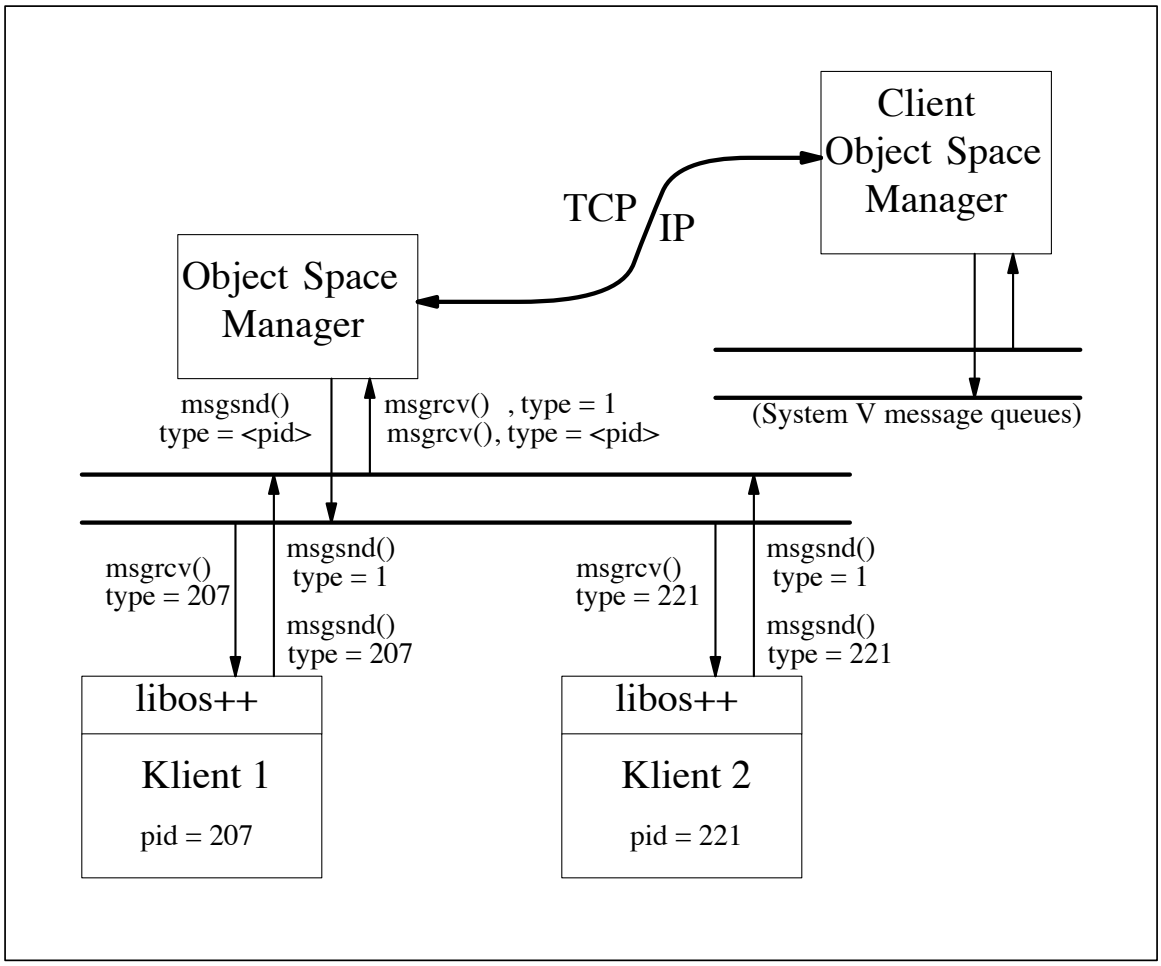
Distributed Type Service

- Classes are identified by unique type id.
- Inheritance sexpressed as relation over those id's.
- simply "counting" of object instantiations is not enough.



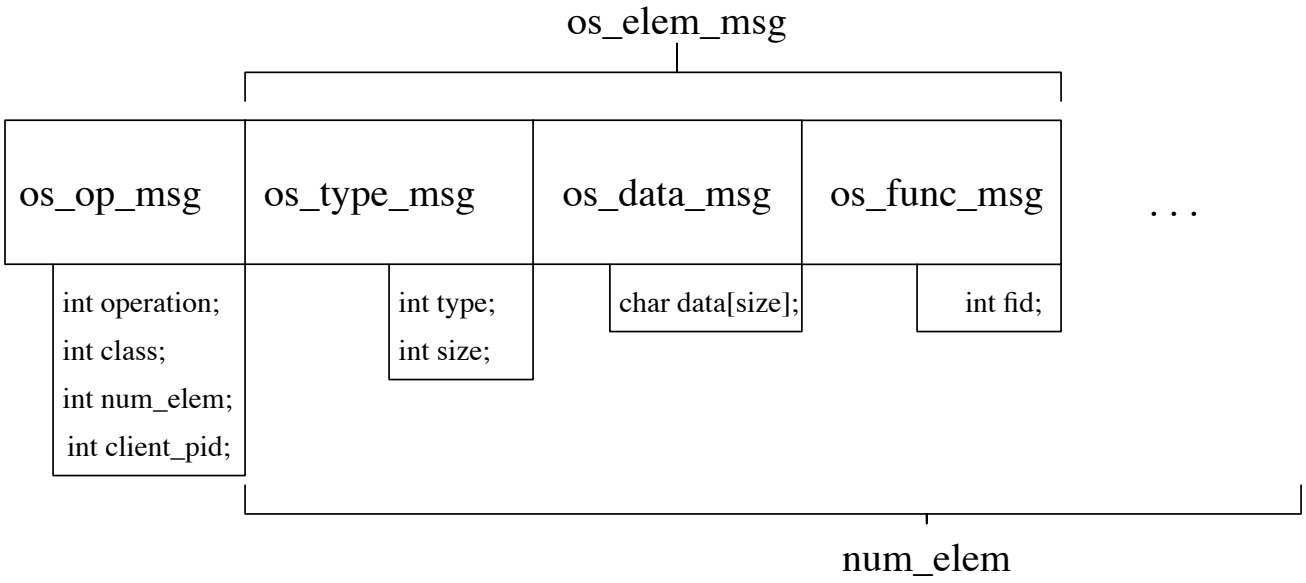
→ pairs `( classname, id )` stored in *Object Space*.

Prototypical implementation of *Object Space*.



- Support for heterogenous environments.
- Local *Object Space* operations are pretty fast (1.05 ms on Sparc 2, DECstation 5000/125).
- A local operation using TCP/IP takes 10 ms.
- An operation issued on a remote node takes 21 ms.

Protocol for communication on *message queues*



- `os_op_msg` is sent with `msg_type==1`, subsequent messages are sent with `msg_type==client_pid`.

  → objects are transmitted contiguously.

- `os_op_msg`, `os_type_msg` and `os_func_msg` are converted into network data format.

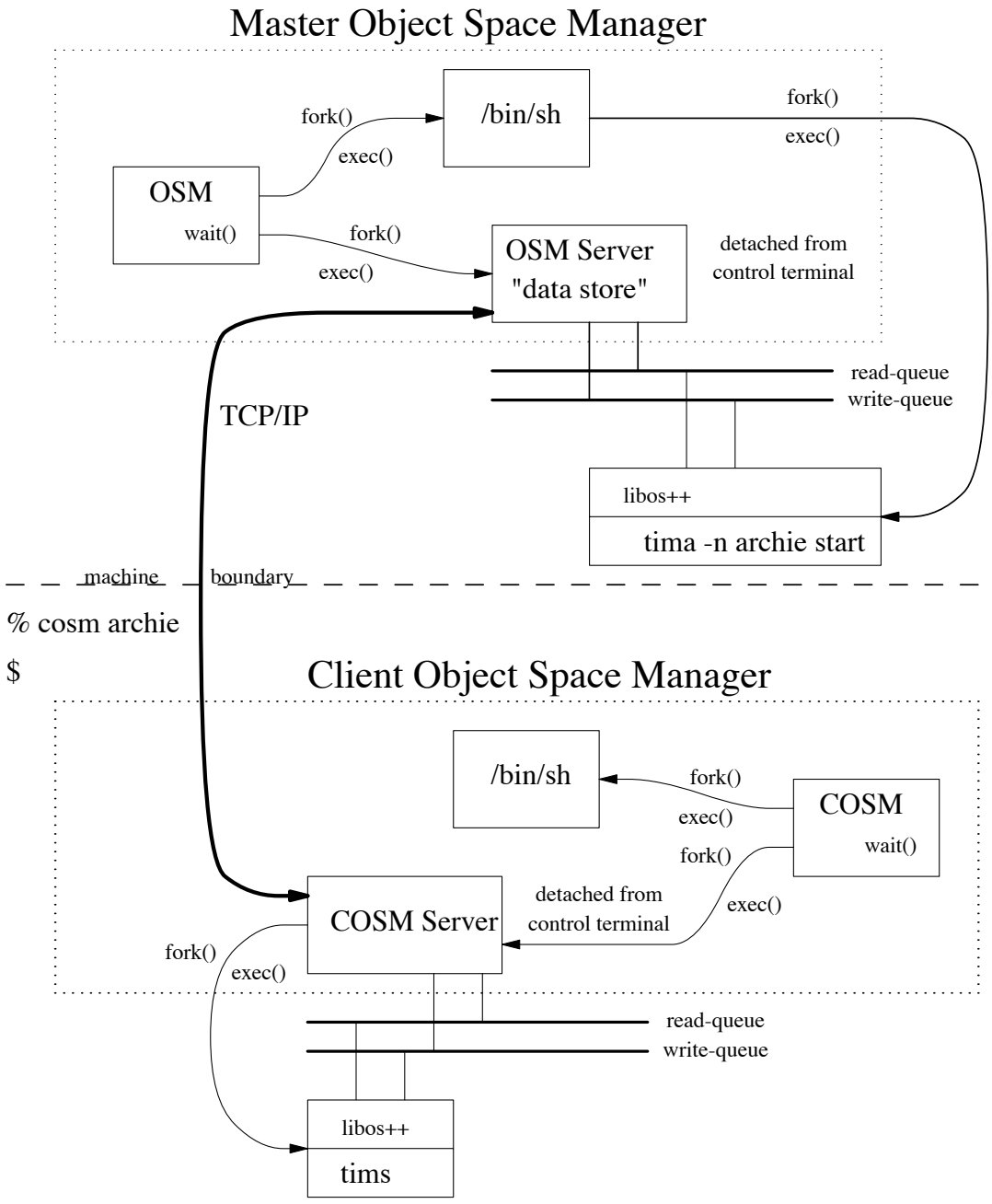
- conversion of `os_data_msg` is up to the programmer.

Launching an Application

% osm telemann

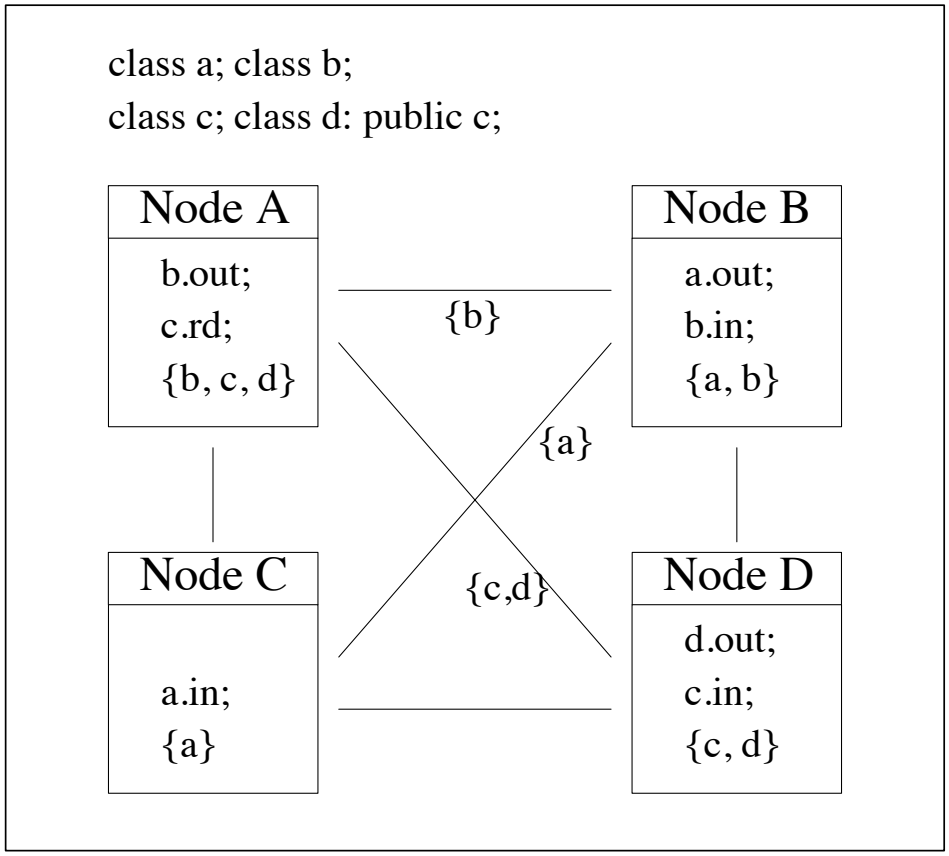
$ tima -n archie start

$

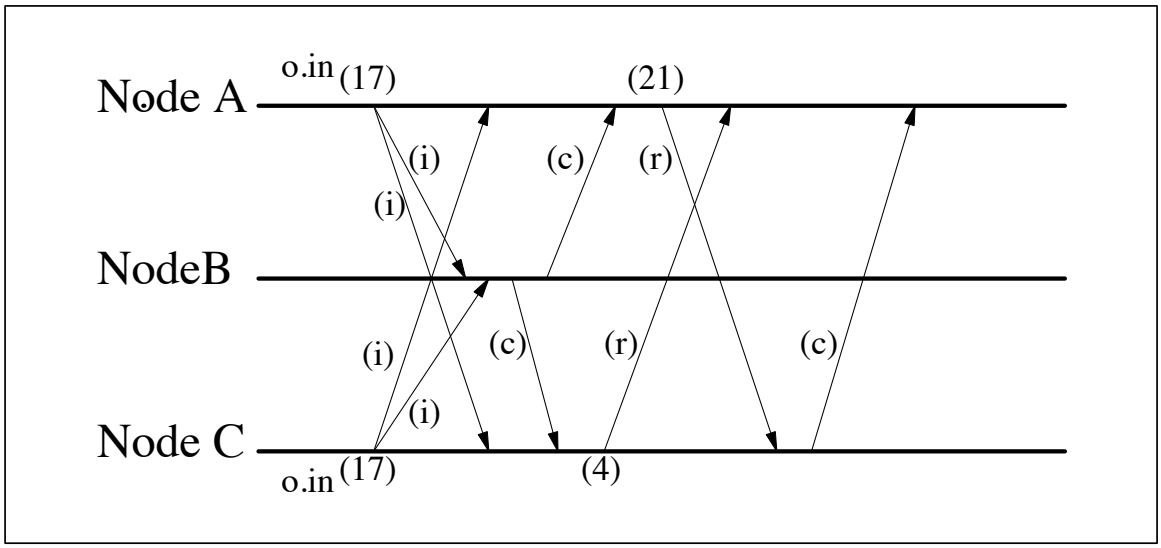


A more efficient implementation

- Replication of objects.
- Distributed type service delivers information about usage of objects of a particular class on a particular node.
- Objects get replicated onto those nodes where instances of the particular class have been used.
- *Object Space Managers* are interconnected to form a complete graph.



- Operation **`out`** works asynchronously. It is not delayed by communication.
- Operation **`rd`** is synchronous. Due to replication of objects that operation may be handled entirely locally.
- Operation **`in`** works synchronous, too. It requires a special protocol to ensure mutual exclusion among *Object Space Managers* when accessing a particular object.
  - Three messages: `initiate-in (i)`, `commit-in (c)` and `retry-in (r)`.
  - Heuristic: messages (i) and (r) get a randomly choosen priority.



How to avoid heuristics within operation **`in`**?

- Associate "owner" with each object

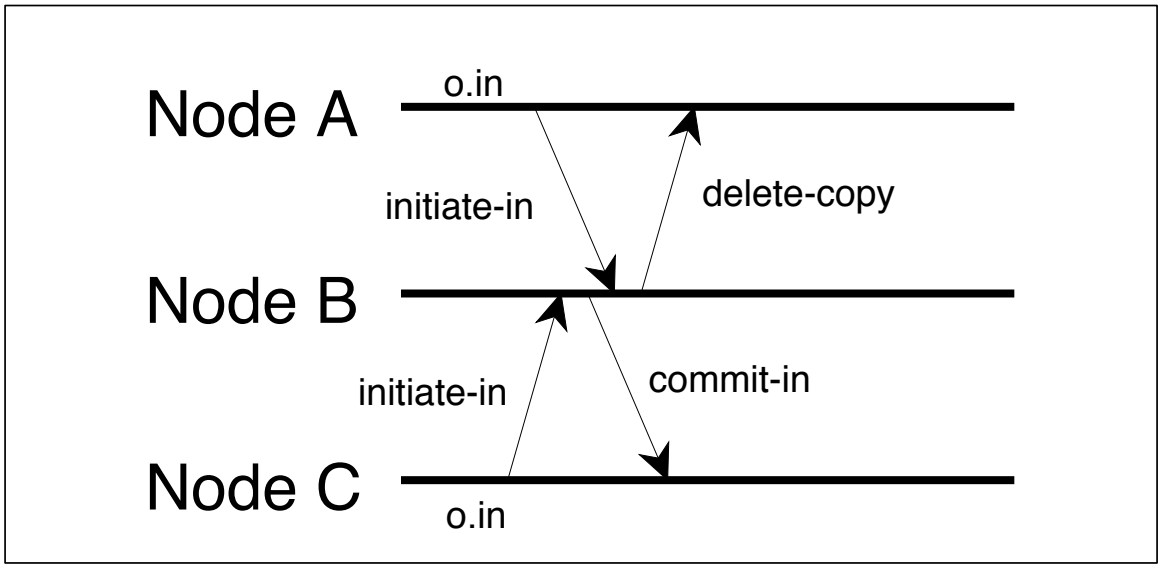
→ the node where an object first appeared.

- Owner decides which node gets object on a race.

→ message "commit-in" is sent to that node.

- Other holder of copies receive "delete-copy" messages.
- Holder of an object's copies elect new "owner" if a node crashes.

→ Operation **`in`** needs exactly two TCP/IP messages.



Acceleration of operation `in`

- *optimistic asynchrony*:
  result of operation **`in`** is returned to an application before all holders of copies of the particular object have sent their `commit-in` message.
- Delay through *message queue* communication only.
- Transaction mechanism allows to set a client process back if its *Object Space Manager* fails when negotiating for an object.
- When trying to perform the next *Object Space* a process is delayed until all `commit-in` messages concerning the previous operation **`in`** have arrived.

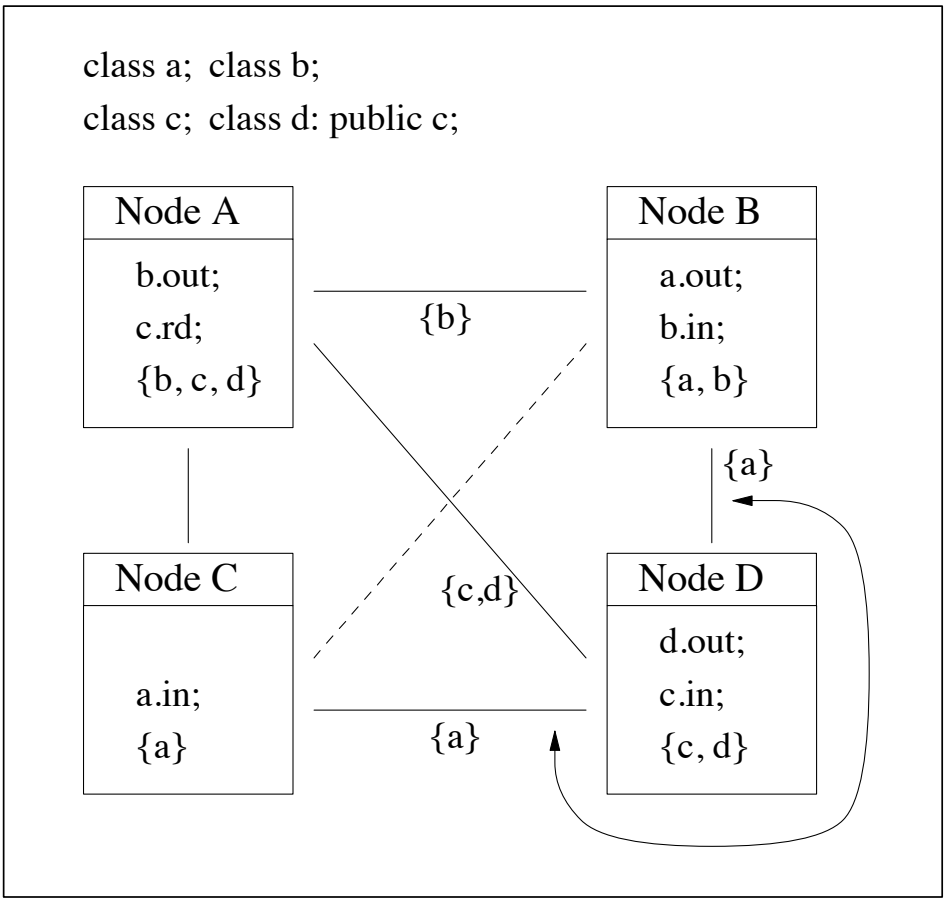
Simple implementation: **`soft-in`**

— `libos++` contains hidden reference for an object already returned to the client.

— Programmer may support a notification function which is called if the result of operation **`in`** changes afterwards.

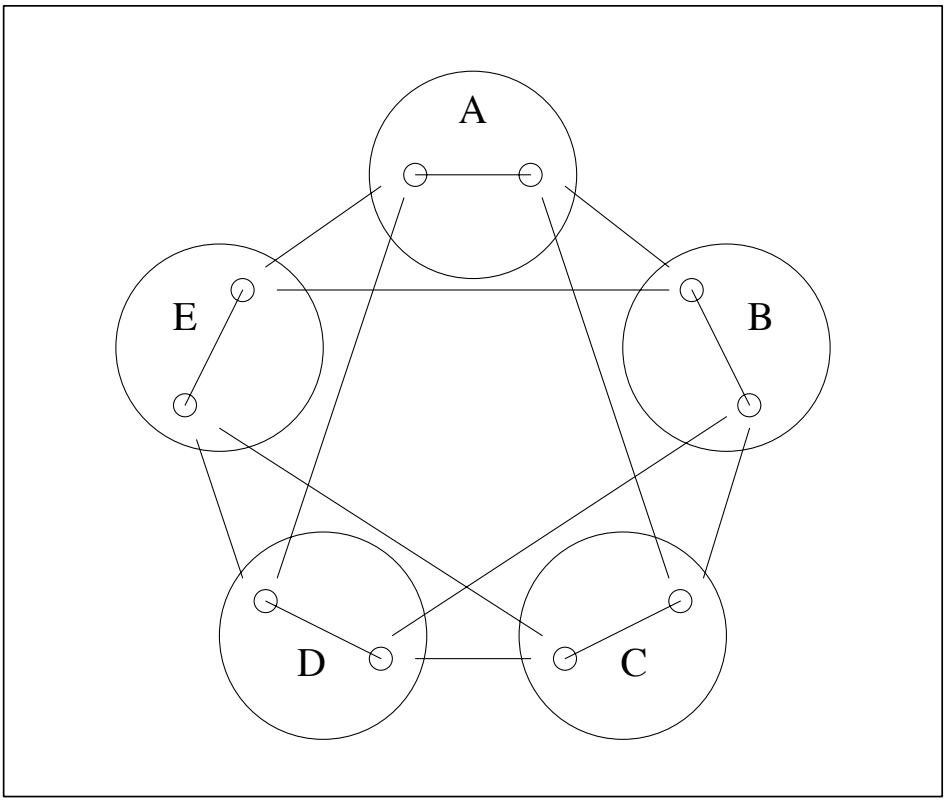
Optimistic asynchrony allows execution of all *Object Space* operations with the costs of fast local communication.

Reconfiguration, Fault-Tolerance



- Open distributed systems — components come and go.
- TCP/IP — when trying to access a broken connection `read()` returns 0.
- Periodical attempts to reconnect.

Other network topologies



- Number of connections per process is limited.
- Moore graph: number of nodes approaches

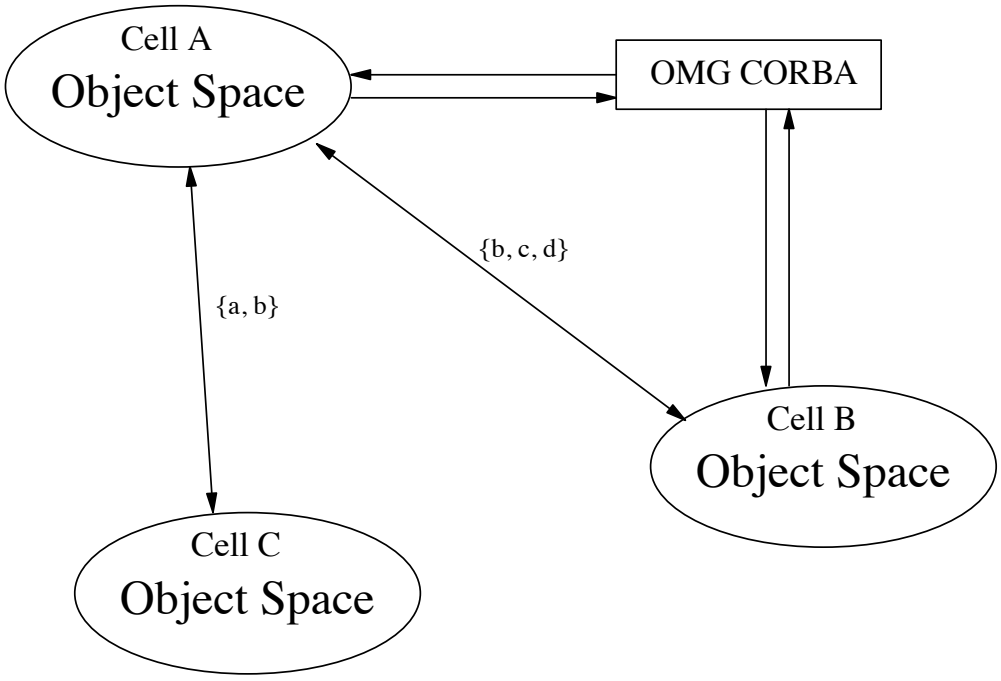
$$\frac{(d^k(d-1)^k - 2)}{d-2}$$

  `d` - maximal elementary path.
  `k` - in-out degree of a node.

- highest possible number of nodes for a given path length.
- new routing algorithms?

Large Scale Systems

- “light” and “heavy weight” classes of objects.
- “light weight” objects are accessible within its cell only.
- “heavy weight” objects can move between cells.



- Connections between cells managed by CORBA.

Conclusions

- Decoupled style of communication is well suited for open distributed systems.
- Objects allow for definition of communication protocols.
- Some concepts of UNIX environment — remote signals — have to be included into *Object Space*.
- Replication and “optimistic asynchrony” promise to speed up operations.
- Future research has to be done for large scale distributed systems.

Future Research Directions

- Parallel Computing in Distributed Environments
- Modern workstations + fast networking technology (ATM) available today.
- Less expensive than special parallel architectures.

New paradigms for programming and memory management in parallel systems pursued!

- *Shared Objects Memory*
  - — objects instead of memory pages
  - — data access vs. synchronization access: weak consistency
  - → avoidance of consistency related communication
- encapsulation of synchronization operations in classes

→ sequential consistency model for ‘‘user’’ of shared objects.

- class library (SOM) + message-based runtime system

- Microkernel-based: CMU's MACH

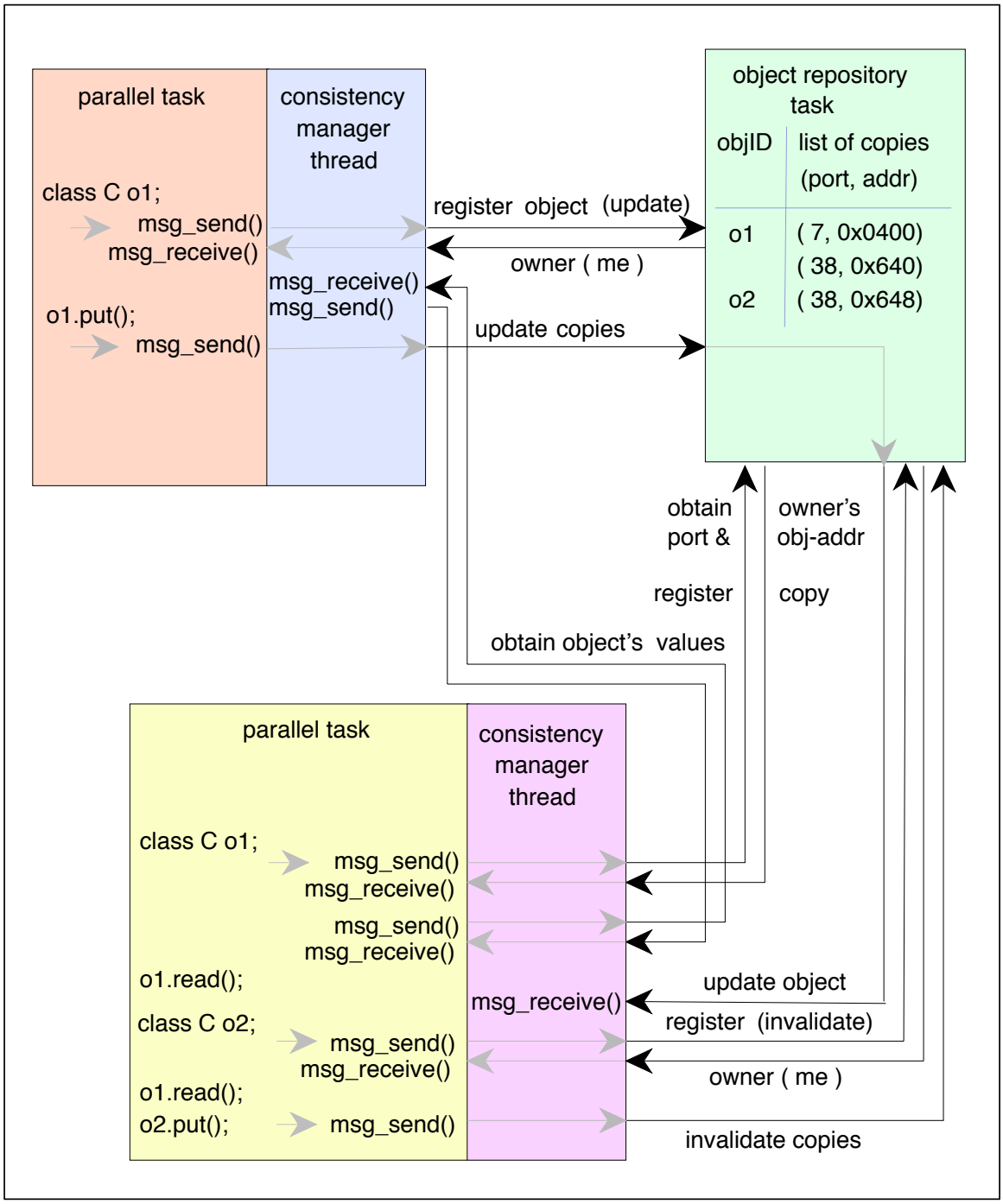
→ Network-transparent communication via ports

→ Ultimately transition to ATM networking technology

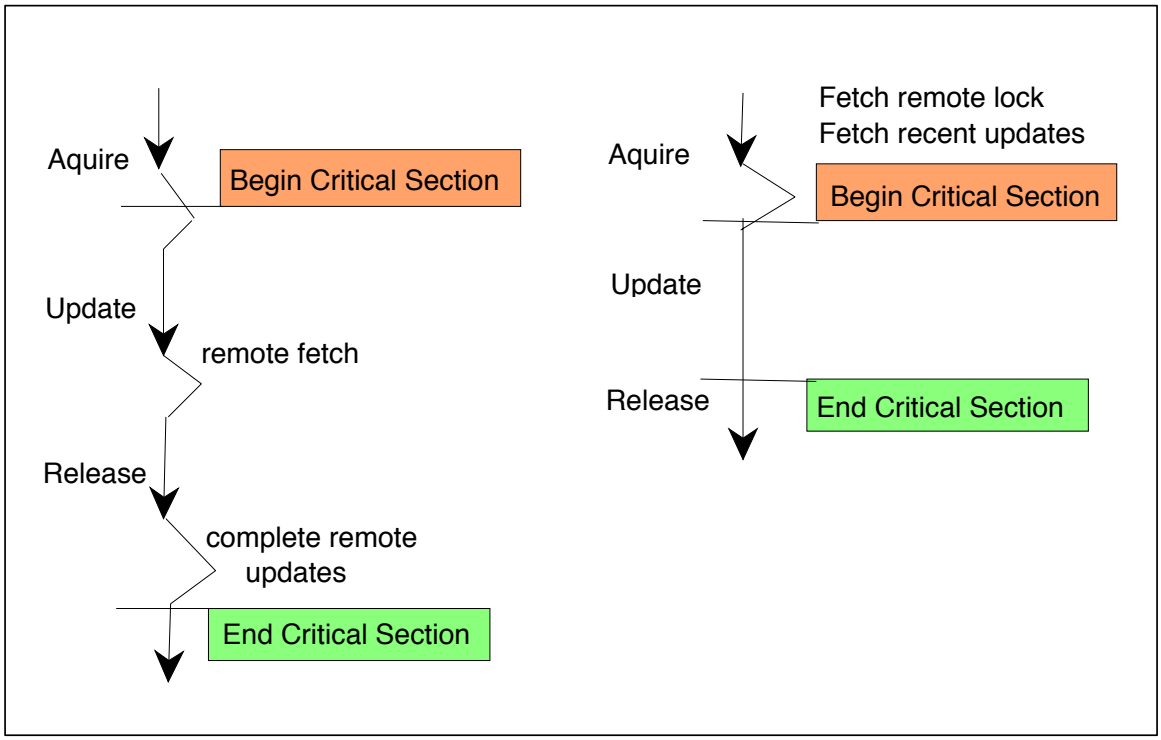
- Tasks in a parallel program are mapped onto multi-threaded Mach tasks.

- C++ base classes provide several consistency models

- Support for object migration & placement


parallel task
consistency manager thread
class C o1;
msg_send()
msg_receive()
o1.put();
msg_send()
msg_receive()
msg_send()
register object (update)
owner (me)
update copies
object repository task
objID
list of copies (port, addr)
o1
(7, 0x0400)
(38, 0x640)
o2
(38, 0x648)
obtain port &
register
owner's obj-addr
copy
obtain object's values
parallel task
consistency manager thread
class C o1;
msg_send()
msg_receive()
msg_send()
msg_receive()
o1.read();
class C o2;
msg_send()
msg_receive()
o1.read();
o2.put();
msg_send()
msg_receive()
update object
register (invalidate)
owner (me)
invalidate copies

Behaviour of two weakly consistency models



Example class: shared counter

```
class counter : public ENTRY_CONS {
        int _i;
public:
        counter( int val ) { _i = val;
            if (! valid()) validate_object(); }
        virtual int size_of()
                    { return sizeof( *this ); }
        virtual void* assign( void* p )
                   { *this = *((counter*) p); }

        counter & operator++();
        void show();
};
counter & counter::operator++() {
        aquire_write_lock(); _i ++;
        release_lock(); return * this;
}
void counter::show() {
        aquire_read_lock();
        printf("obj: %d, val: %d\n", oID(), _i );
        release_lock();
}
```

References

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

[Carriero et al. 86] N.Carriero, D.Gelernter, J.Leichter ; *Distributed data structures in Linda*; Thirteenth ACM Symposium on Principles of Programming Languages, ACM SIGPLAN and SIGACT, Williamsburg, Virginia, January 1986.

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

[Lipovski/Malek 87] G.J.Lipovski, M.Malek; *Parallel Computing — Theory and Comparisons*; John Wiley & Sons, Inc., 1987.

[Polze 93] 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.

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