University of Los Andes
Engineering School
Center of Distributed Systems and Microelectronics (CEMISID)

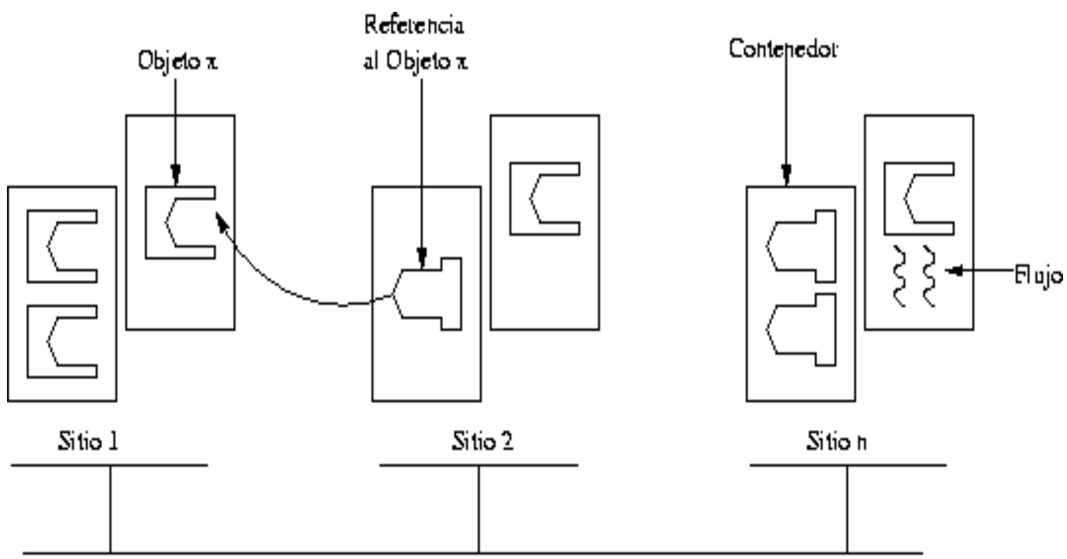
Techniques for Locating Mobile Objects

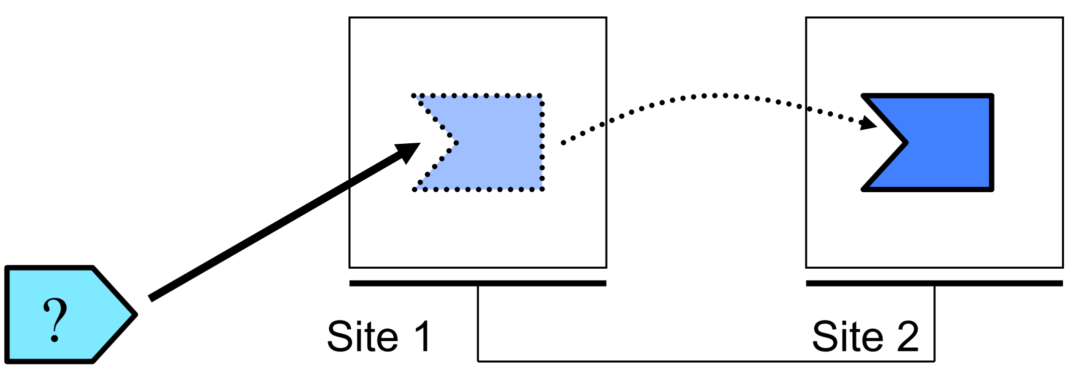
Andrés Arcia – Leandro León

amoret@ula.ve - lrleon@ula.ve

Mérida, january 2002.

Object migration

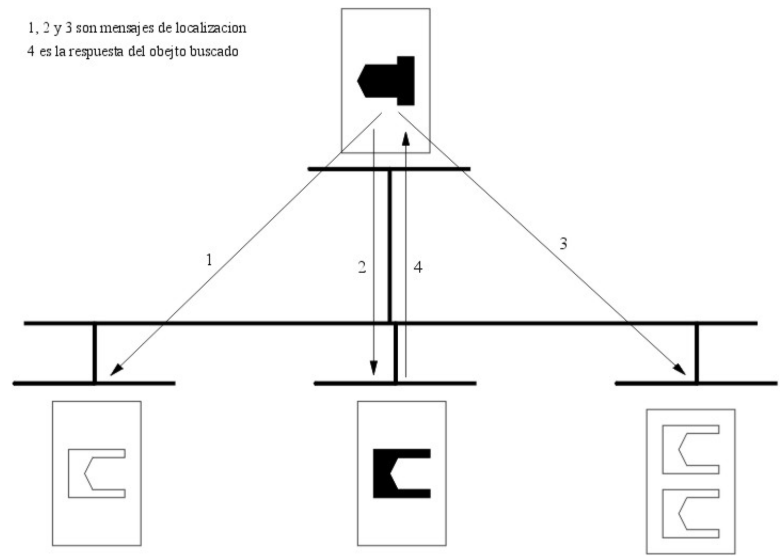




- Consecuences:
  - Invalid references.
  - Invocation failure.
  - Invalid references should be somehow updated.

Introduction

- Broadcasting is the simplest solution for updating invalid references.
  - Broadcast the new address, then migrate.
  - Broadcast for the address before doing an invocation.

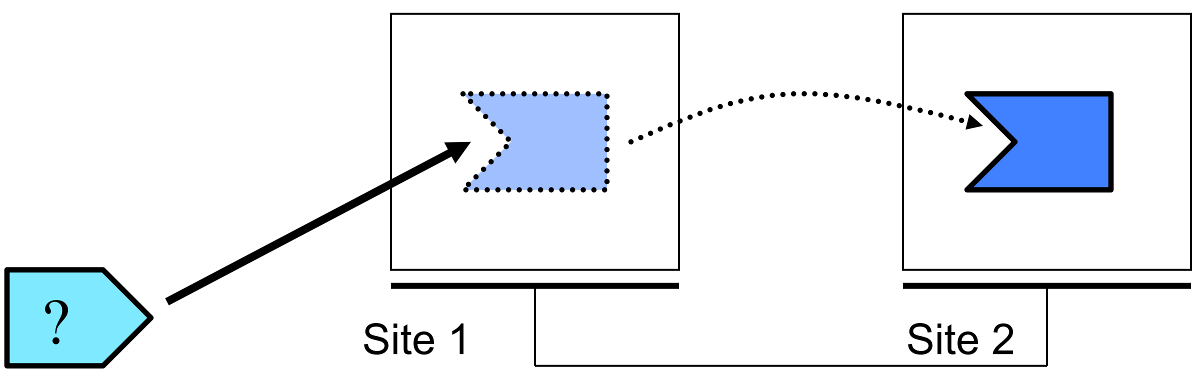


Test and update

A test for every reference is done just before an invocation, so that a client can check whether the server remains in the same place. This method has been used by Emerald, SOS, Amber.

Problems:

- All invocations do the test.
- It works well for low scale systems.

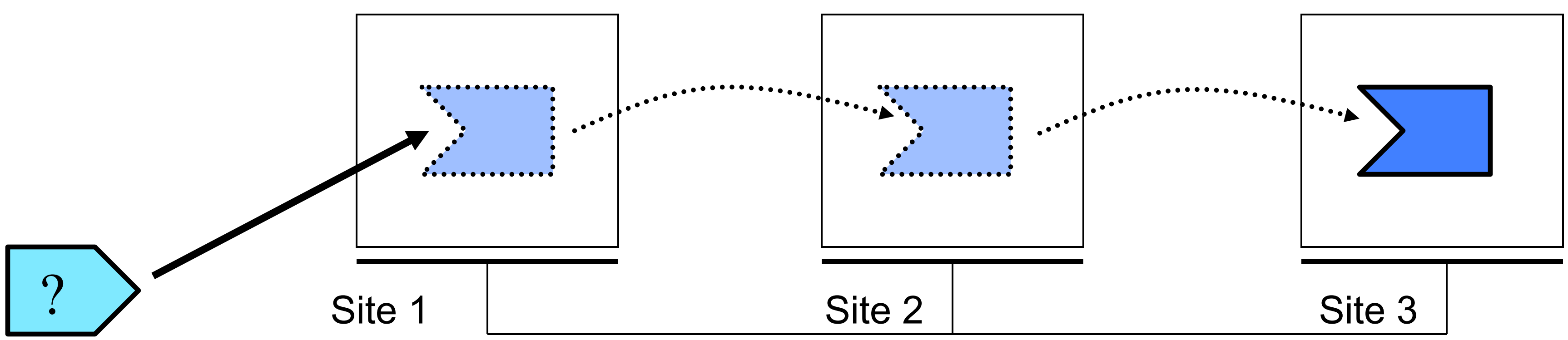


Forward addressing

A link is left in the migration source site and it points to the migration target site. This method is widely used by the following systems: Emerald, Guide 2, Demos/MP, Galaxy, DC++.

Problems:

- Garbage collection.
- It has to be a part of the migration protocol.



General problems

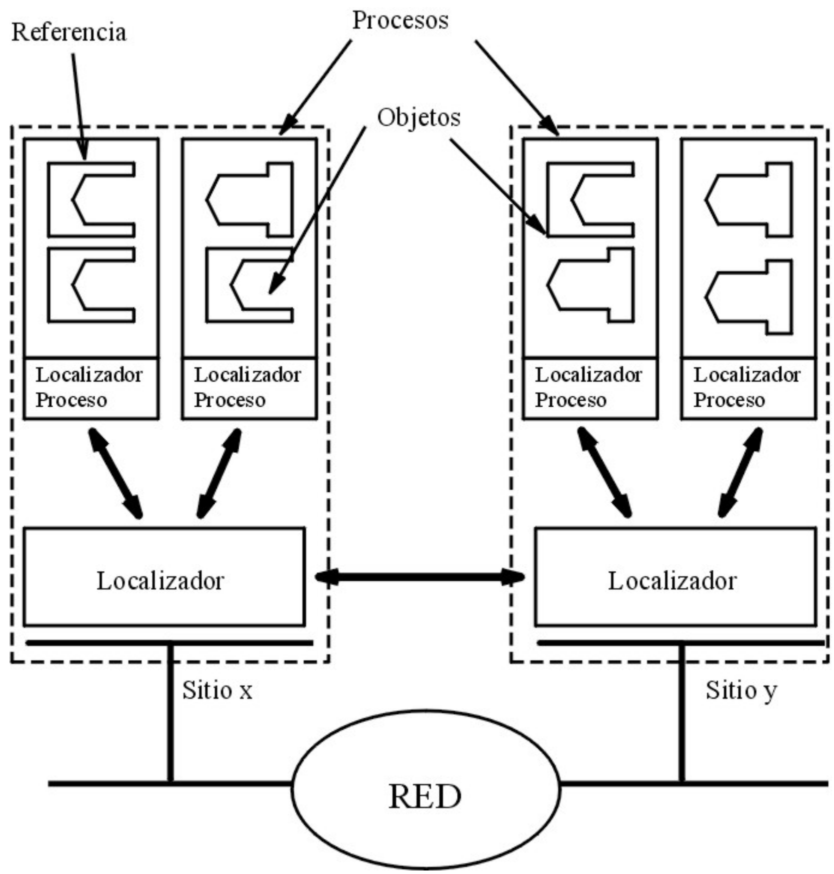
- Fail tolerance
  - Hard to detect failures.
  - Difficulty for solving a failure recovering.
- Performance
  - Garbage collection.
  - The higher the scale the lower the performance (more memory, cpu and messages).

Objectives

- *Versatility*: The system is parametric and allows to choose among different techniques.
- *Portability:* It is widely portable among different platforms and o.s.
    - It is IP-based.
    - It has been programmed in standard C++.
    - It is POSIX compliant.
- *Distribution:* It is completely distributed.
- Fail tolerant:
    - It has been proposed a fail-recovery protocol.
    - It has redundant techniques for locating objects.

General service architecture

- A locator kernel per site
- A run-time library per process.
- Centralized interface, but distributed and cooperative service



General service architecture

- Locator guidelines:
  - It knows the location of every object and process.
  - It keeps the necessary information for distributed location.
  - It manages local and distributed object location.
  - It can intercept incoming and outgoing invocations.
  - It cooperates with the fail-recovering protocol.
- Runtime-locator guidelines:
  - It keeps information of all objects and its methods
  - It checks and dispatches incoming invocations
  - It manages object migration.
  - It cooperates with the fail-recovering protocol.

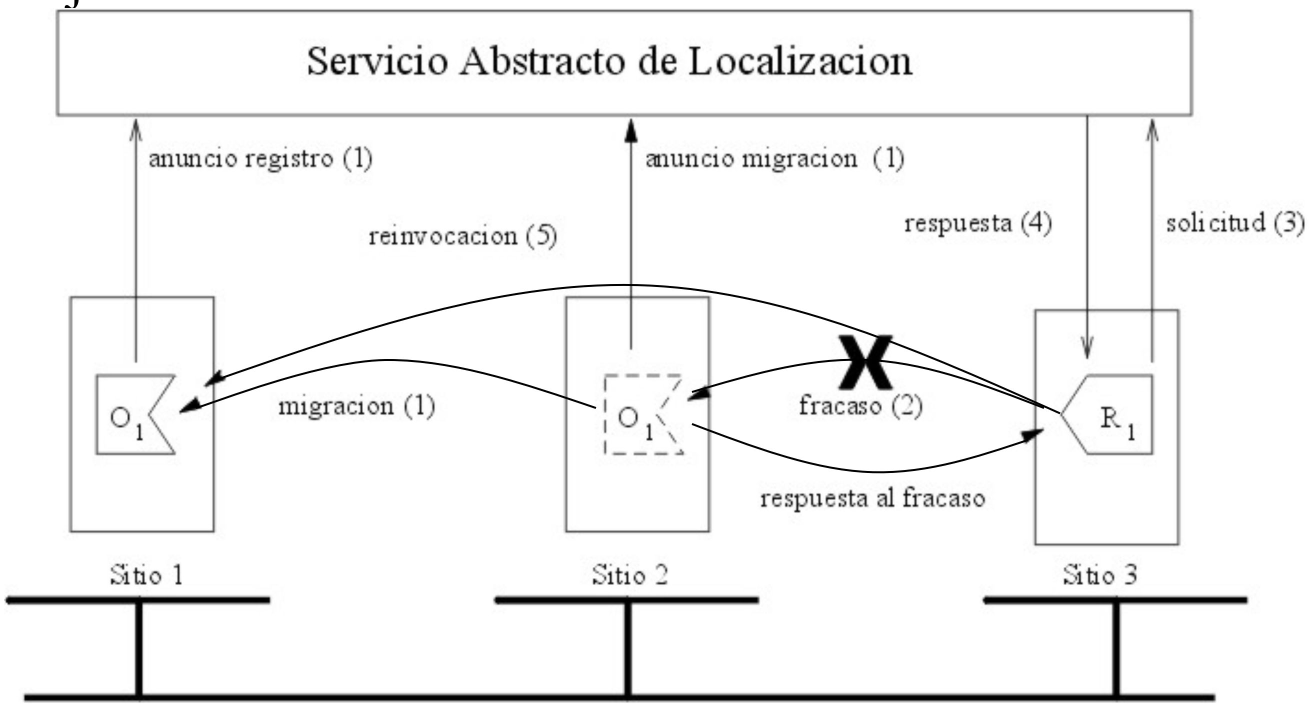
Location techniques

- Caching
- Prefetching
- Piggybacking in inter-locator messages
- Broadcasting by stages.

Reference update by invocation failure

The searching process take place once an invocation has failed. The failure can be known through one of these three events:

- The object was not found
- There exist a more recent reference
- The object has been deleted



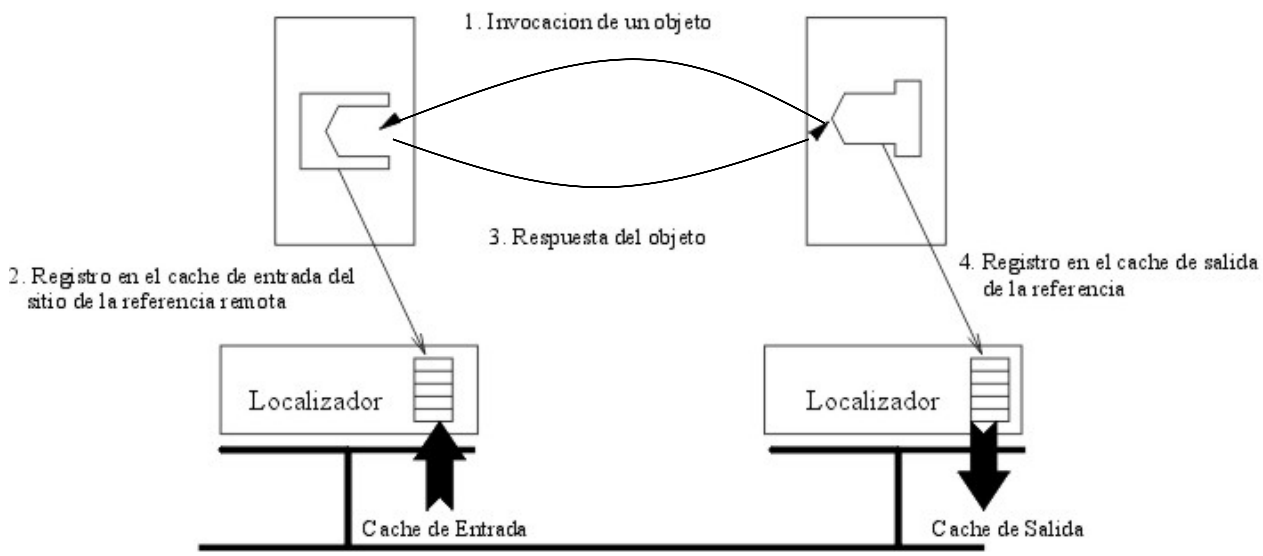
Caching

The caching consist of saving recent object locations. Those locations are obtained through searching or gotten from update messages.

- Types of caches:
  - ***Input***: It keeps information about invokers from other places.
  - ***Output***: It keeps information about the objects being invoked.
  - ***Migration***: It keeps information about object migration.
  - ***New references***: It keeps information about unused references.
  - ***Deletions***: It keeps deleted objects.

Input / output cache

- **Input cache**: A record for this cache is of the form <O,s_o,t>. O is the object ID, s_o is the source site of the reference, and t is a logical timestamp.
- **Output cache**: A record of this cache is of the form <O,s_d,t>. O y t are the same as those described in the input cache. s_d is the site where O is.



Cache updating

The good performance of caching is reached through:

- Updated caches: migrations have to be cached, and kept as updated as possible.
- Prefetching: it is done through inter-locator messages.
- Recent information is handed through piggybacked messages.

Prefetching I

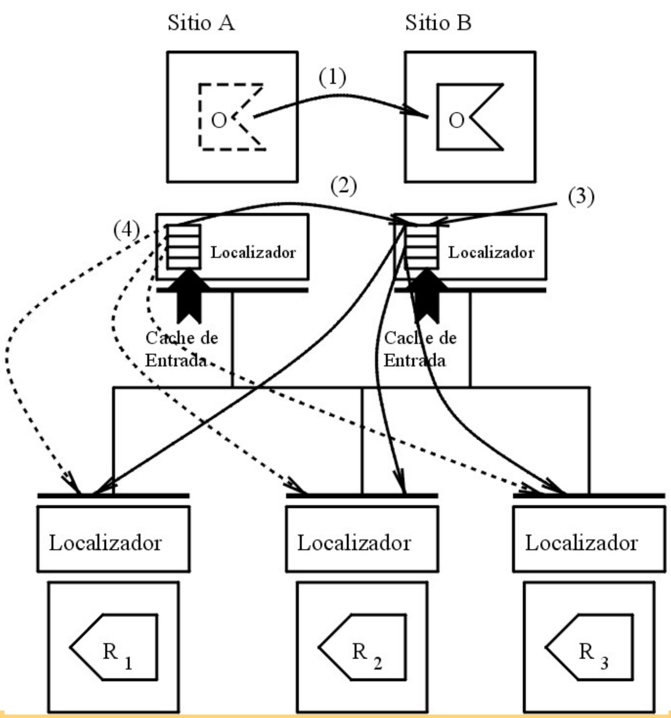
This technique is used for going ahead of invocation failure. So, through the use of prefetching, it is possible to update a reference before it fails.

Types of prefetching:

– Prefetching from the source site of the migration.
– Prefetching from the target site of the migration.
– Prefetching with the output cache.

Prefetching II

- Prefetching from the source site of the migration:
  - ***With the input cache***: Look for the records of the migrating objects and notify the new address to the source sites.
  - ***With the migration cache***: If an invocation to a migrated object arrives, it should be answered the address being kept is this cache.



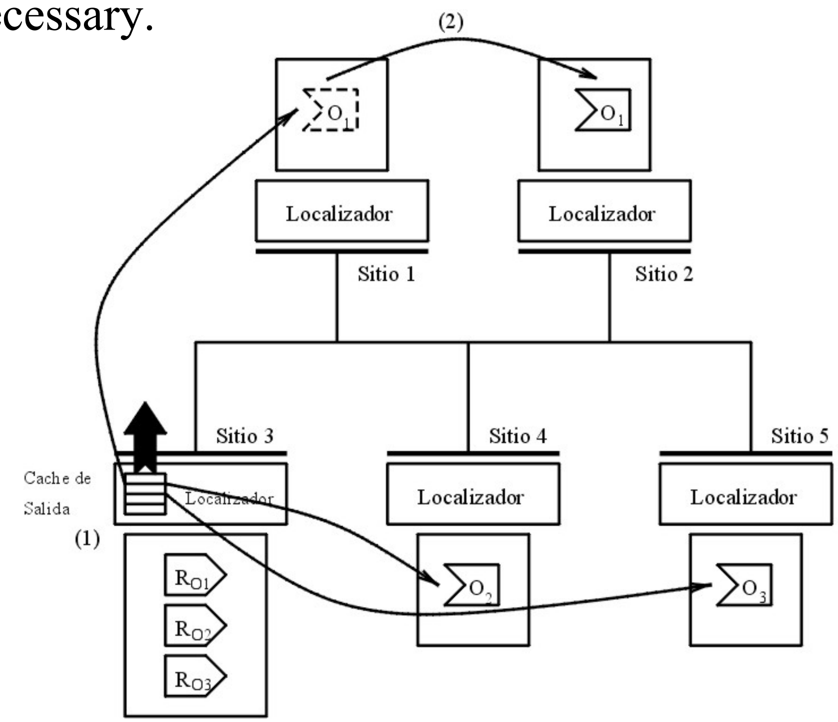
Prefetching III

- Prefetching from the target site of the migration:
  - ***With the input cache***: The records with the migrating object have to be copied to the target-site input cache. So the prefetching with the input cache can be applied again.
  - ***With the output cache***: Check the references in the output cache and convert them to local references if necessary.

- Prefetching with the output cache:

  - It consist of periodicaly check the validity of the references. If a reference is found to be invalid, actions for updating can be started before time.

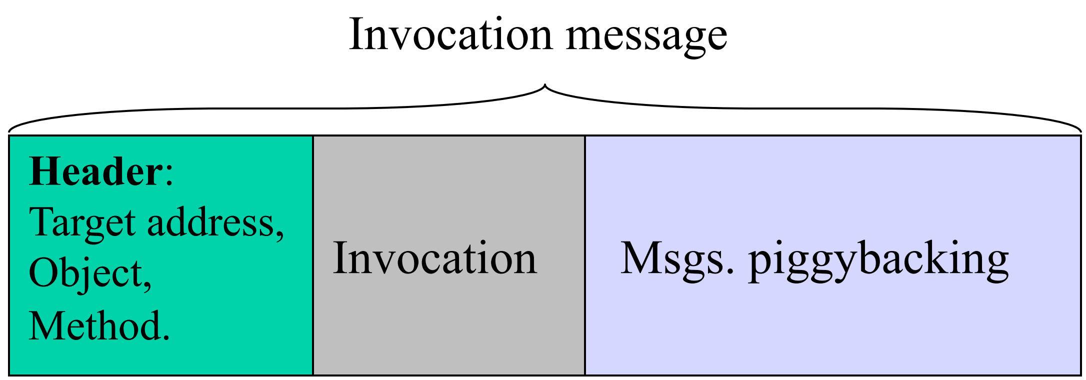
  -New references are transparently used from this cache.



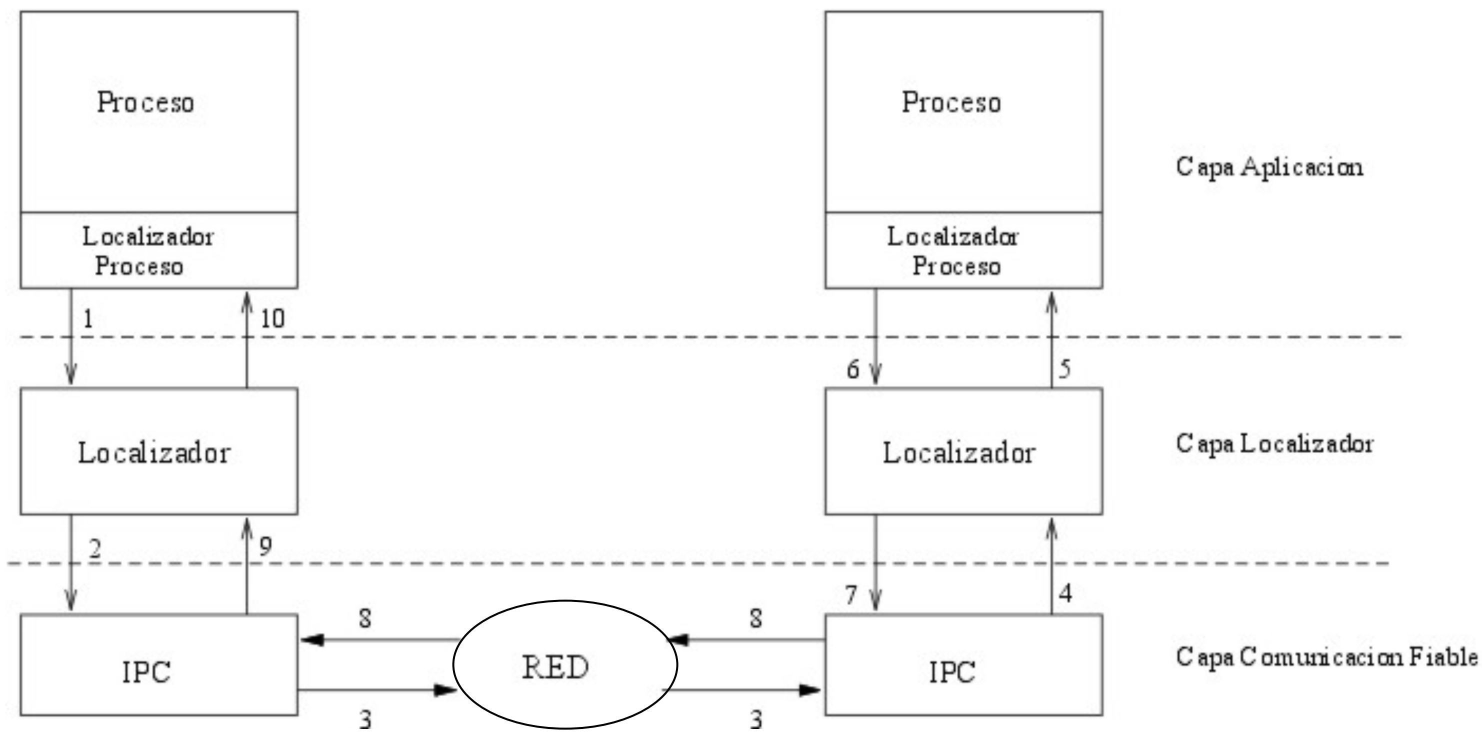
Piggybacking

Given a inter-locator message (invocation, update message, etc):

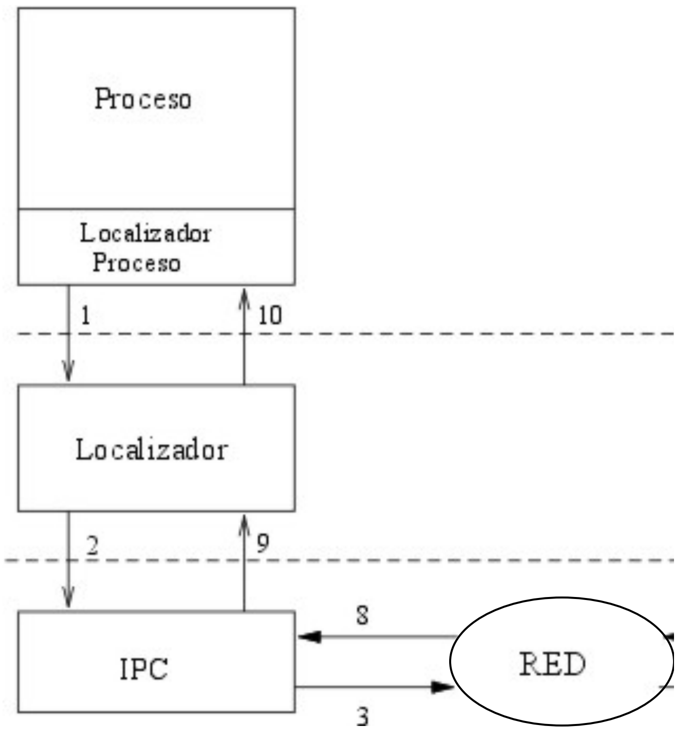
- Additional messages can be added.
- MTU size can be completed with some of those messages.



The invocation system I



The invocation system: client side



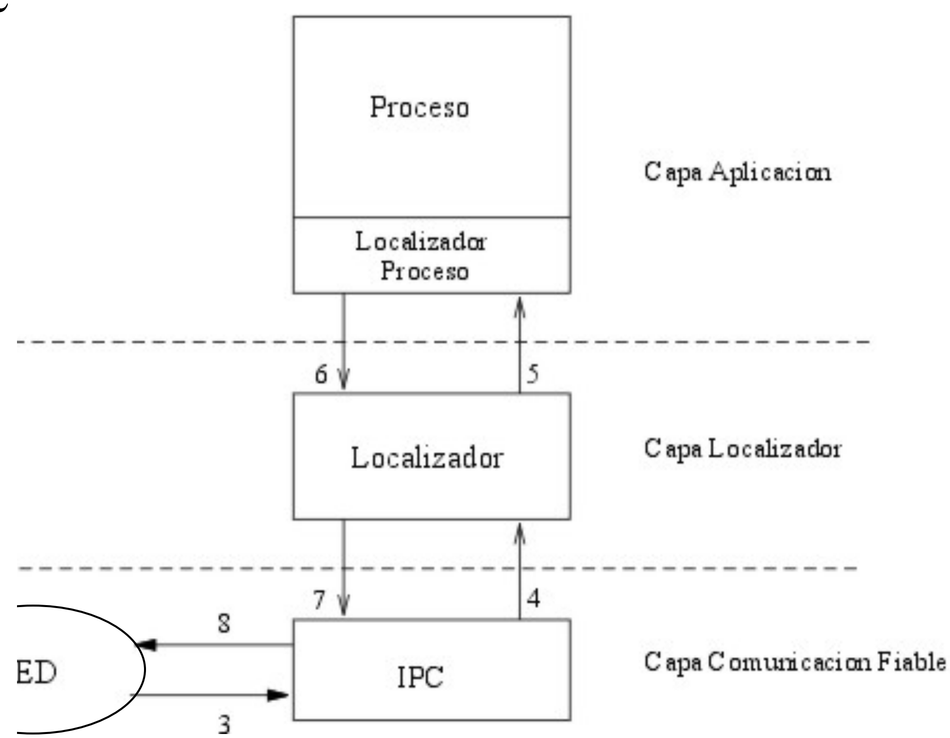
1. Invocation system call to the locator.
2. Passing of a network message to the IPC system.
3. Reliable sending of the IPC request.

...

8. Reliable sending of the IPC reply for the request of step 3.
9. Reply handing to the locator system by IPC.
10. Upcall that unblocks the waiting process.

The invocation system: server side

3. Reliable sending of the request by the IPC system.
4. Pass of the invocation request message from the IPC to the locator.
5. Upcall that unblocks the waiting server process that contains the requiered object.
6. The invoked object does a reply.
7. Pass of the invocation reply message from the locator to the IPC system.
8. Reliable sending of a reply by the IPC system.



Piggybacking messages

1. ***Object deletion***: This message is used when an object has been deleted.
2. ***Object finding***: This message indicates that an object is being sought or a stale reference has to be updated.
3. ***More recent reference announce***: This message propagates an updated reference. It could be used when a new object appears or when an object migrates.

Piggybacking traffic control

Piggyback messages spend resources (memory, cpu, network, etc.)

⇒ ***Number of piggybacks messages has to be limited***

⇒ ***piggybacks messages propagation has to be limited***

Techniques:

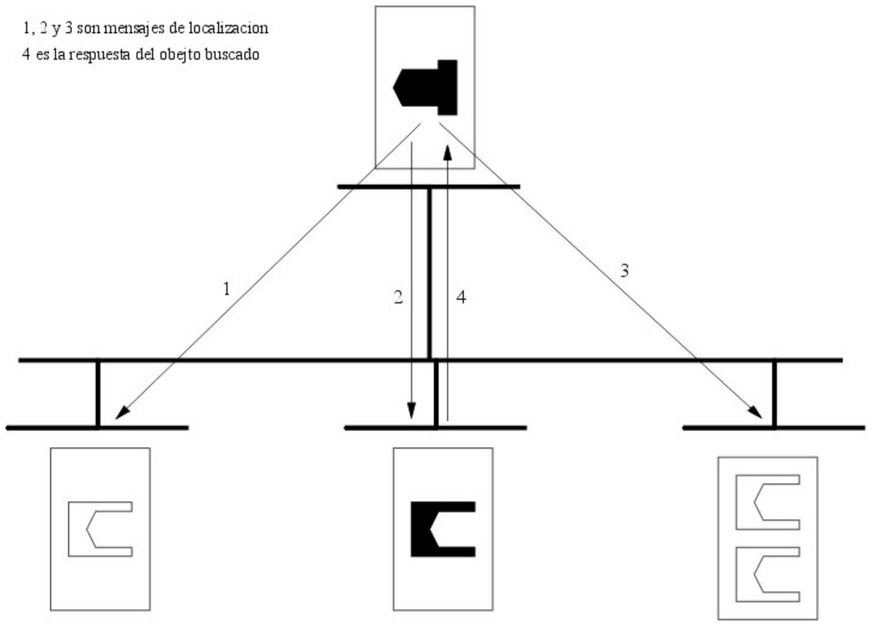
- *Priorities*
- *Site graph, frequency graph*
- *Logical timestamps*
- *Physical time*
- *Limit in the number of messages*

Broadcasting by stages

There are two kinds of broadcasting:

– Unreliable broadcasting: low cost.

– Reliable broadcasting: high cost.

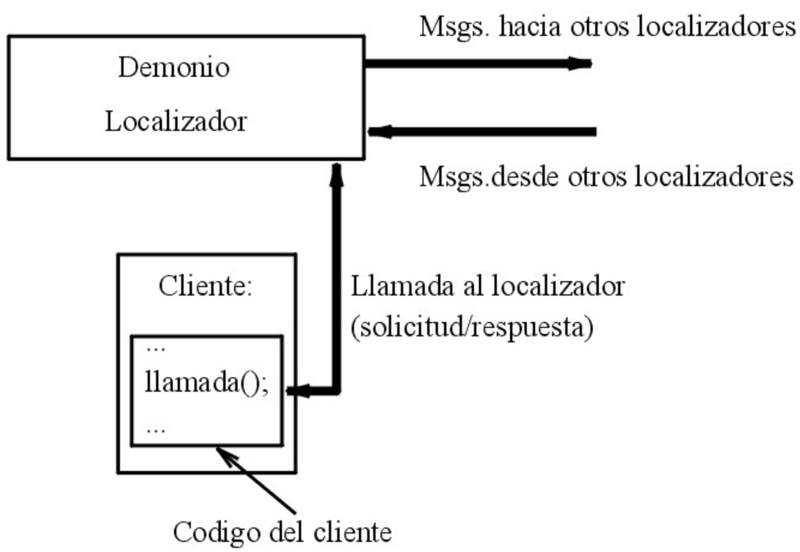


Broadcasting protocol

- Weak broadcasting (unreliable)
    1. Make a weak broadcasting asking for the object's owner.
    2. If the owner has not been found, then make a weak broadcasting asking for the information in caches.
- Strong broadcasting (reliable)
    1. Make a strong broadcasting asking for the object's owner.
    2. If the owner has not been found, make a broadcasting ordering to start the inconsistency-recovery protocol.
    3. If yet the object has not been found, then the object is taken as if it were deleted.

Location system interface

- It is composed by a set of functions that we are going to call locator system calls. This interface offers a centralized view of the system.
- Locator system calls are made from clients.
- The locator clients have to link a run-time library.
- It is used C++ exception system.



Register calls

– Process registering calls

```
void register_prc(Process_Id &)
     throw(Duplicated)
void unregister_prc(const Process_Id &)
     throw(NotFound, RefusedService)
```

– Objects registering calls

```
void register_obj(Object_Id &, const Process_Id &)
    throw(Duplicated, NotFound)
void unregister_obj(const Object_Id &)
     throw(NotFound)
```

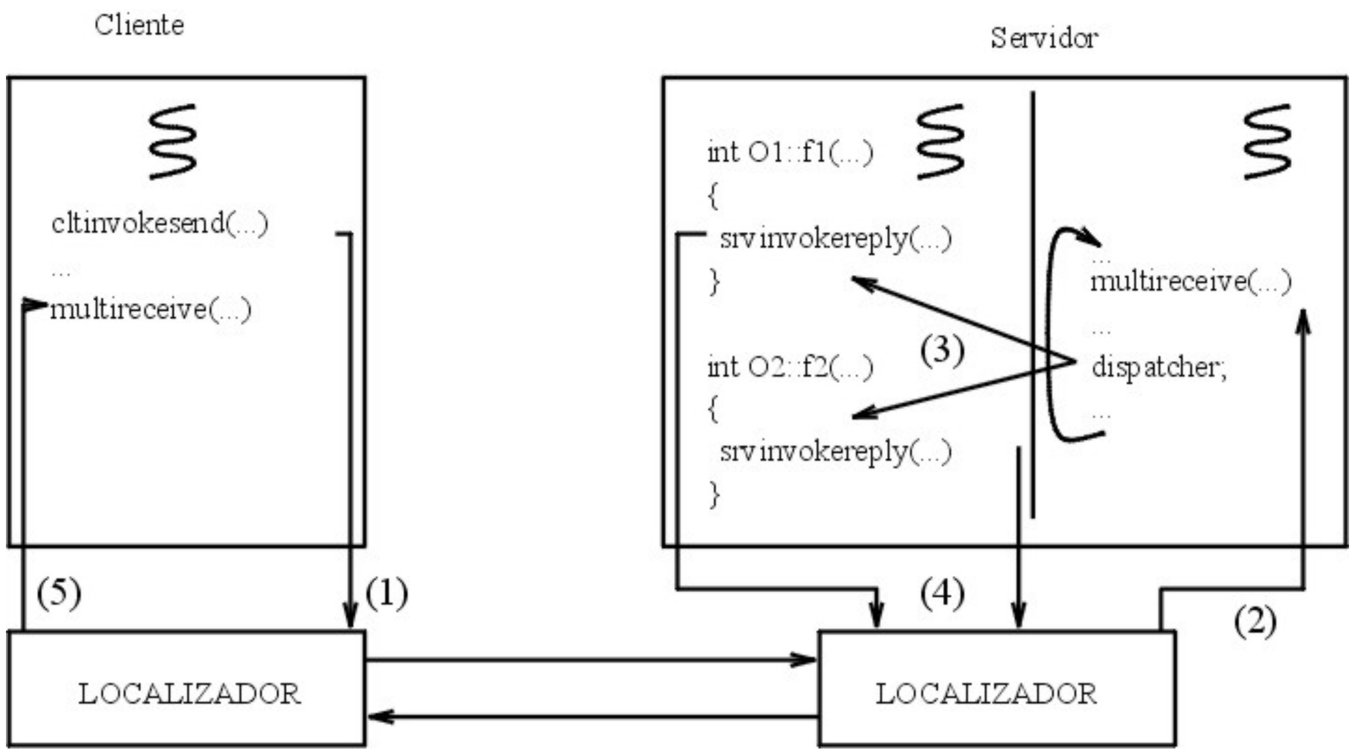
Invocation calls

```
Message_Id multi_receive(Binding & binding,
                         const Process_Id & receiving_process_id,
                         void * data,
                         size_t & data_size,
                         Reception_Type & message_type)
throw (NotFound, ObjectDead, RecentBinding)

Message_Id clt_invoke_send(Binding &, const void *, const size_t)
throw (ObjectDead)

void srv_invoke_reply(const Message_Id &, const Binding &,
                      const Process_Id &, const void *, const size_t)
throw ()
```

Invocation calls II



Migration calls

– Object migration calls

```
void src_unreg_mig_obj(const Object_Id &, const Site_Id &)
    throw(NotFound, ObjectBusy)
void tgt_reg_mig_obj(const Object_Id &, const Process_Id &,
                     const Logical_Timestamp)
    throw(NotFound, Duplicated)
```

– Process migration calls

```
void src_unreg_mig_prc(const Process_Id &, const Site_Id &)
    throw(NotFound)
void tgt_reg_mig_prc(const Process_Id &)
    throw(Duplicated)
```

Location calls

```
Locator strong_locate(const Object_Id &)
      throw (ObjectDead)

Locator weak_locate(const Object_Id &)
      throw (NotFound, ObjectDead)

void implicit_locate(const Binding &)
      throw ()

void test_location(Locator &)
      throw (NotFound, ObjectDead)

void ping(int number_of_entries, const Cache_Update_Policy policy)
      throw (NotFound)
```

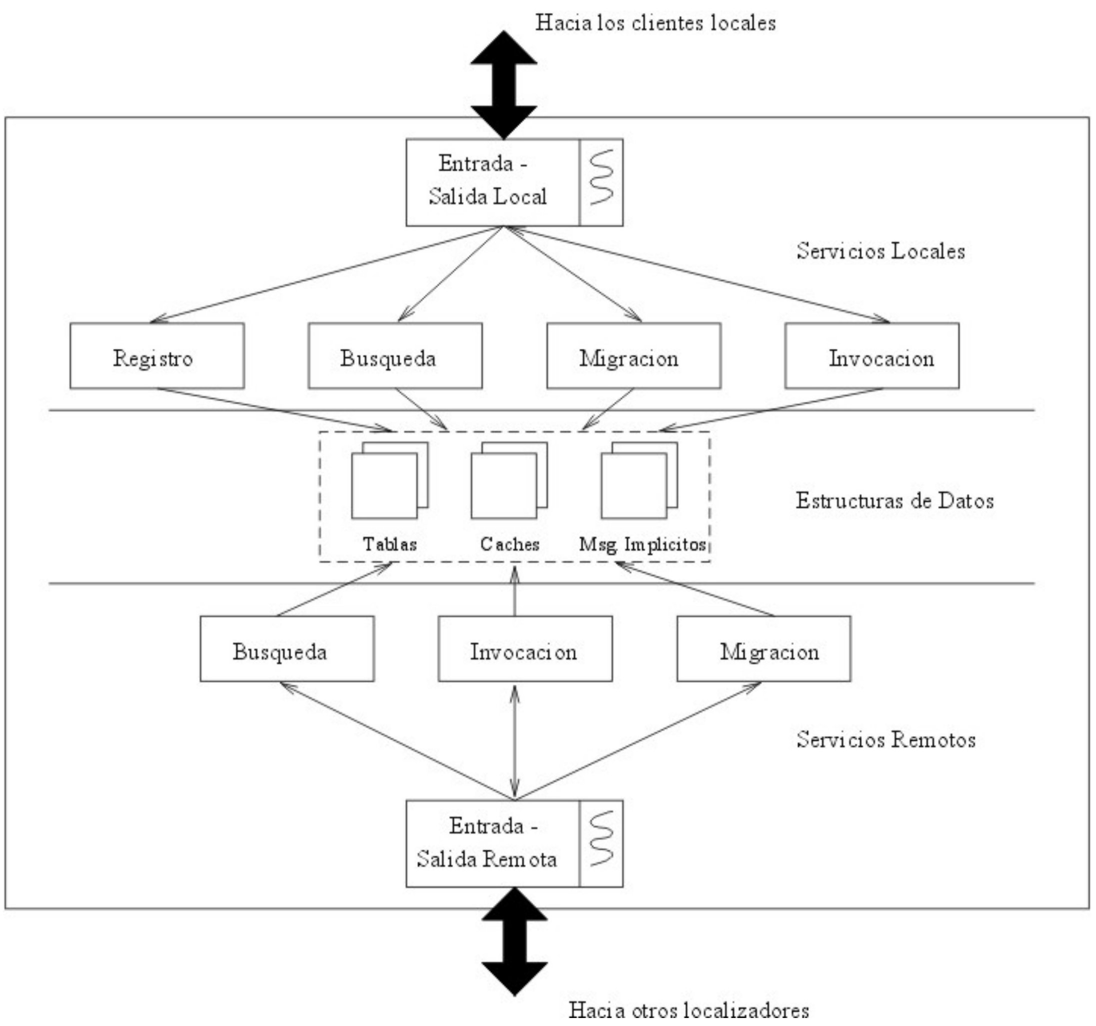
An example: a server code.

```
# include "locator_calls.H"
// usage: server <number of iterations> <number of objects>
# define RECEPTION_BUFFER_SIZE 4096
int main(int argc, char ** argv)
{
  bootstrap_services();
  ...
  Site_Id   this_site(INVALID_SITE_ID);
  get_site_id(this_site);
  Object_Id this_object;
  Process_Id this_process;
  // REGISTRATION STAGE
  register_prc(this_process);
  register_obj(this_object, this_process);
  // BINDING FOR INVOCATIONS
  Binding binding;
  Message_Id msg_id;
  Reception_Type reception_type;
  size_t reception_size;
```

An example: a server code.

```
for (int services = -1;
       (services < n_times - 1) || (n_times == -1);
       n_times==-1 ? n_times = -1 : services++)
    {
      reception_size = RECEPTION_BUFFER_SIZE;
      msg_id = multi_receive(binding, this_process,
                    reception_buffer, reception_size,
                    reception_type);
      // processing of the invocation
      srv_invoke_reply(msg_id, binding, this_process,
                    reception_buffer, reception_size);
    }
  unregister_prc(this_process);
return 0;
}
```

Locator architecture



Inter-locator messages

The inter-locator messages announce events between locators. Those events could be invocations, location actions, etc.

The network messages:

- Invocation request
- Invocation reply
- Search for object
- Reference anounce
- Deletion anounce
- Ping for output cache
- Ping answer
- Input cache sending

Performance

- The experiments are simple, low scaled, and do not represent a real OO distribuited application.
- This is a first approach to an evaluation of the system.
- 5 sites, 2 process per site, 5 objects per process and 5 references per process. It summarizes 50 objects and 50 references for the whole system.
- Invocations occurred with probability p.
- Migrations occurred with probability q, where $p+q=1$.

Performance

p	q	Success	Techniques
0.9	0.1	99.2%	Caching, Prefetching
0.7	0.3	94.38%	Caching, Prefetching
0.7	0.3	97.3%	Caching, Prefetching, Piggybacking ☺

Conclusions

- Location and invocations of mobile objects can be done through a simple interface.
- The code is POSIX compliant and it is programmed is standard C++. So the system is portable between different Unix versions.
- The system is scalable. This is an inherited property from the system architecture and the techniques for updating reference.
- The system has been programmed with a high degree of cohesion and low coupling. So, this means that the system can be enhanced and modifed more easily.

Future perspectives

- Forward addressing in transport layer
- Fast objects capture
- Analitical and simulation models for cache updating.
- Higher scale experiments

Thank you.