# Sparing for an OO CASE: A Transparency approach for building Fault Tolerant Systems

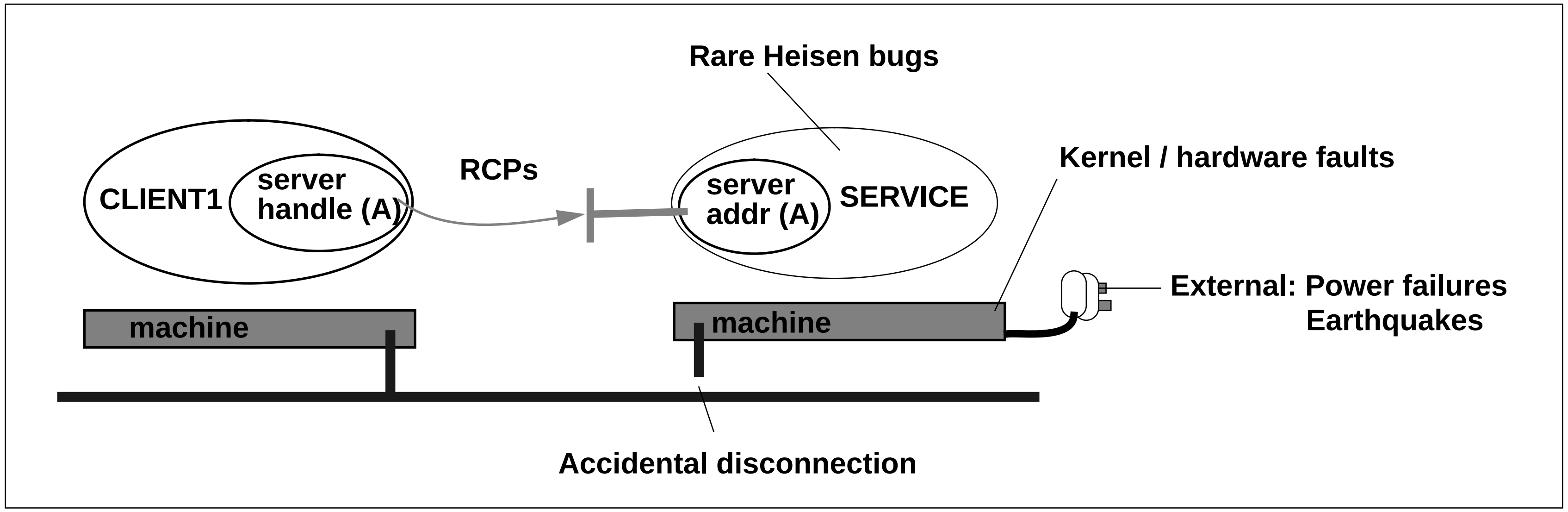
Yoko Chung
Distributed Systems and Interactive Broadband
Advanced Technology Centre
BNR Europe Ltd
01279 403686 (Y.K.Chung@bnr.co.uk)

## Availability within Telecommunications

- High availability is an important non-functional requirement in the telecommunication systems.
- General requirement for 99.999% availability.
- Core switching & call management components tend to use specialised hardware for fault tolerance.
- Example - synchronous dual processing.
- Specialised hardware expensive but necessary in switches as operations require the fast throughput.
- Call management can be made cheaper.
- Use off the shelf hardware and software fault tolerance mechanisms.

## Faults in Distributed Systems

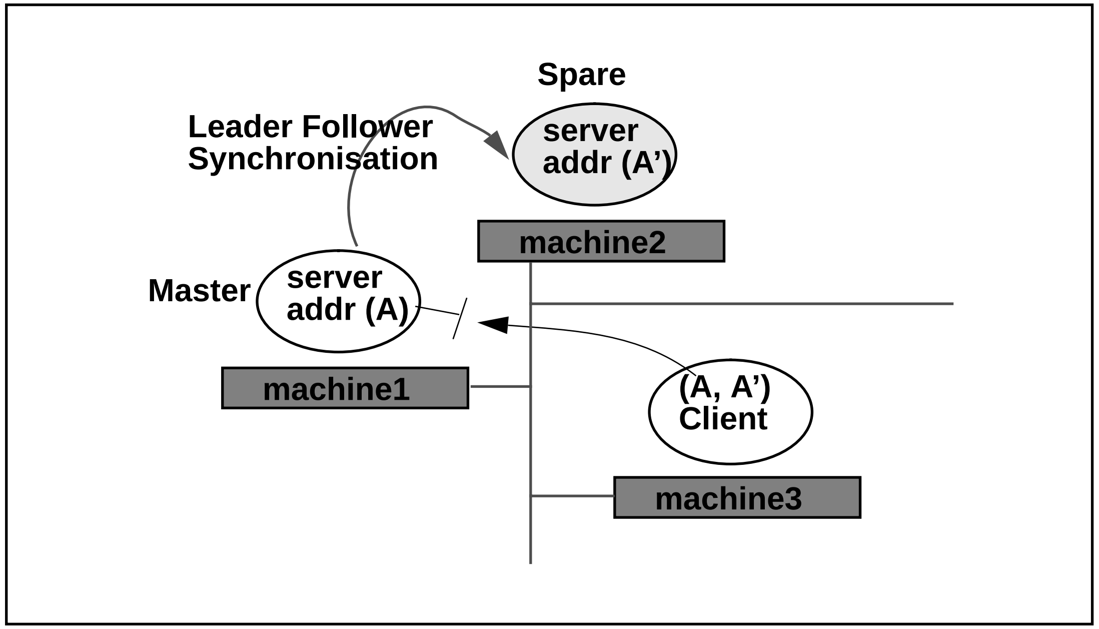
- Lots of faults can disrupt interaction between distributed objects.



- Types of faults include:
  - Hardware - failure of devices.
  - Software - Transient bugs disappear when re-examined.
  - Operation - accidents may occur.
  - Environmental - earthquakes.

## Introduction to Sparing

- **General solution to increase availability of services is through replication.**
- **Several replication schemes available;**
    - **E.g., passive groups, active groups, broadcast protocols.**
- **Sparing is the simplest form of a passive group.**



- **Can be achieved using two off the shelf real time workstations configured with one running as master and the other as spare.**

## Sparing Strategy

**Software advantages:**

- **Hardware option for Fault Tolerance expensive.**
- **Software fault tolerance enables the use of general hardware.**
- **Enables exploitation of the power curve.**
- **Software enables fast turnaround to handle changes in resiliency requirements.**

**Sparing advantages:**

- **Simple and optimal.**
- **Appropriate performance for soft real time systems.**
- **Data consistency and synchronisation flows in one direction.**
- **Provides a way for online version upgrade without loss of availability.**

## Master and Spare Synchronisation

- Sparing involves the following components:
  - master and spare data consistency - recovery and continual update.
  - heartbeats, failure detection and switchover.
  - Transport mechanism between Master and Spare - uses the ONI real time distributed platform which provided a choice of lightweight reliable messages, reliable casts and RPCs.

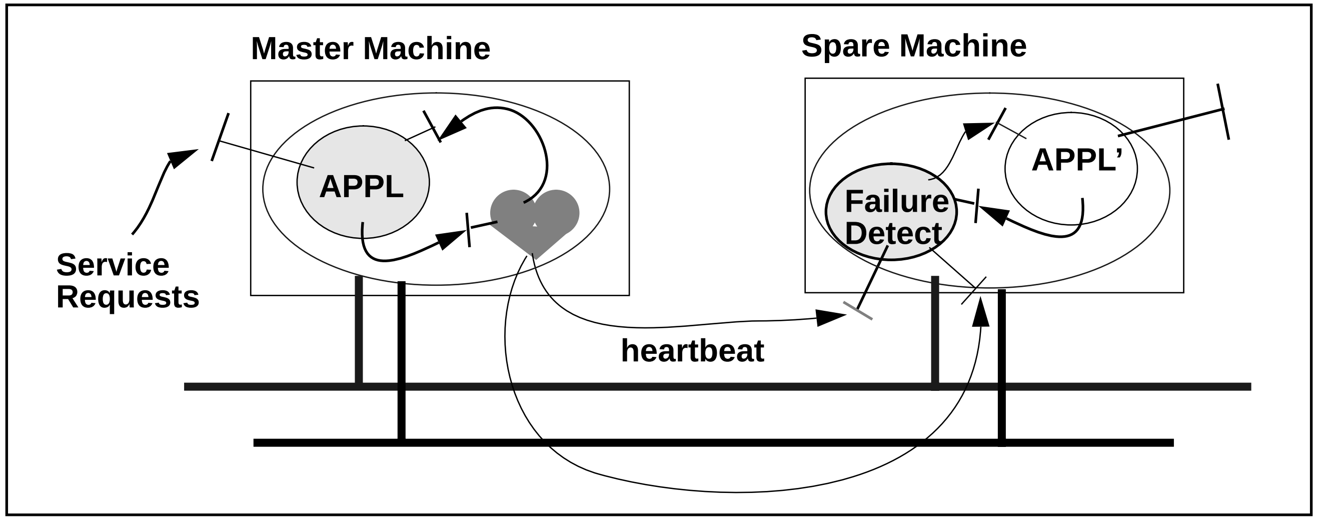
## Recovery

- Resilient objects are given unique ids and are kept track of at both the master and spare.
- Spare maps master objects ids to its corresponding resilient objects.
- Recovery uses 2 phase approach;
  - 1st phase recreates all resilient objects at the spare which are present on the master.
  - 2nd phase sends states of objects to the spare. 2nd phase recovery is interleaved with normal synchronisation from the service updates.
  - 2 phase approach allows pointers to objects to be translated.

## Continuous update/Journalling

- Transactions are required for strict synchronisation. This can be relaxed for more optimistic approach.
- Transactions may not be crucial - each update or *journal* could be atomic.
- 2 Approaches to journalling:
  - State based approach - each journal captures value of object which is then copied by the spare object.
  - Event based approach - each journal captures event which is then replayed on the spare object.
- Hybrid approach taken.
  - Transitional approach used for deterministic events such as object creation and deletion.
  - State based approach for checkpointing of individual objects state.
- Object/data context sparing necessary for the spare to be warm and reduce switchover time -> structured approach to creation/deletion of resilient objects.
- To by pass the state based approach for updating individual state, each spare object was given the ability to replay user defined events on receiving journals.

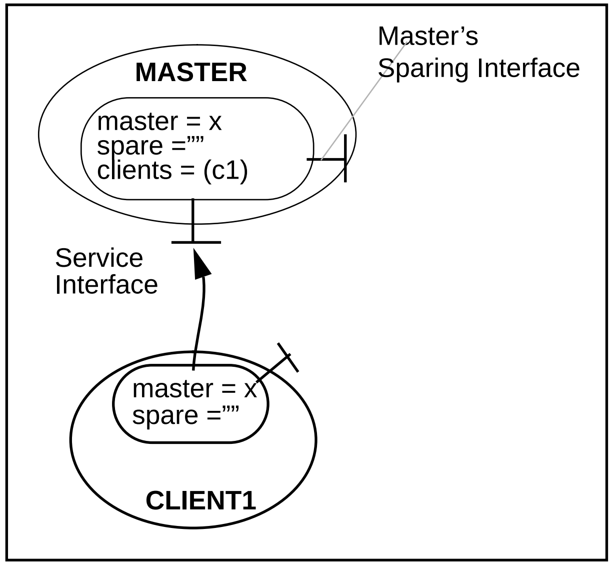
## Heartbeats and Failure Detection



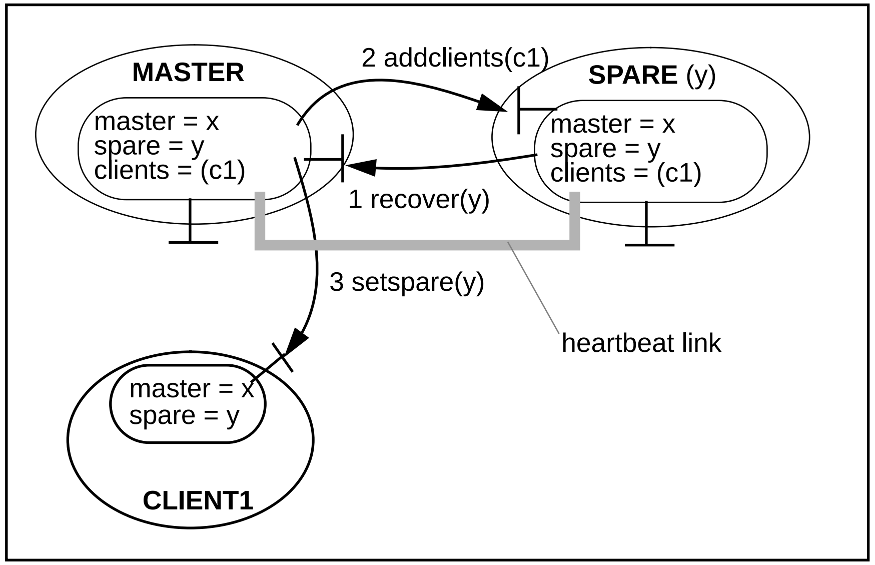
- Doctor and patient analogy.
- Periodic heartbeats are sent from master to spare using ONI reliable casts.
- Failure detection latency dependent on heartbeat priority & send intervals.
- Degree of network fault tolerance achieved through duplication of the heartbeat delivery path.
- Spare initiates switchover on failure detect.

## Spare & Client Switchover Scenario

- **Master failure verification requires third party participation.**
- **Clients need spare and master address information.**
- **Master and spare need to know client addresses because;**
  - **the master will need to tell clients of new spare.**
  - **the spare will need to tell clients to switchover.**

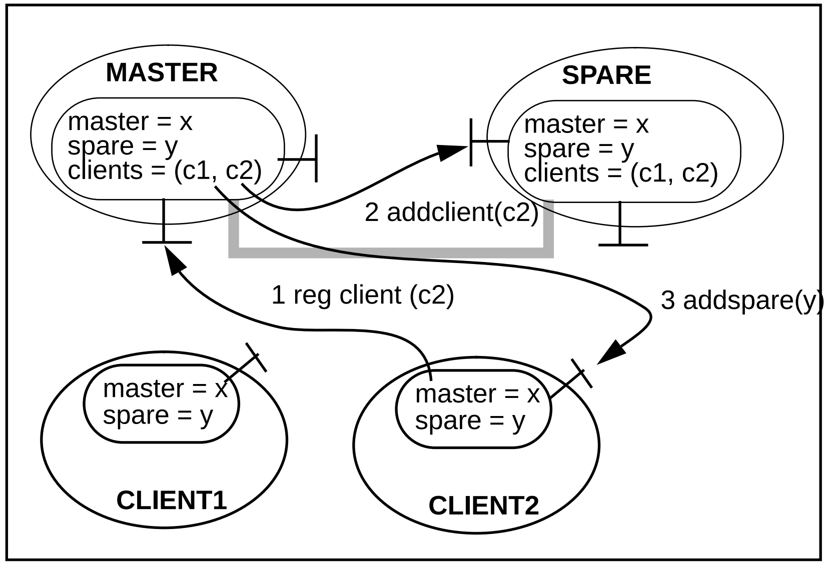


1 Initial scenario has 1 master and 1 client.
Client obtains sparing interface which encapsulates master and spare interfaces - client only sees one and messages are only sent to the master.

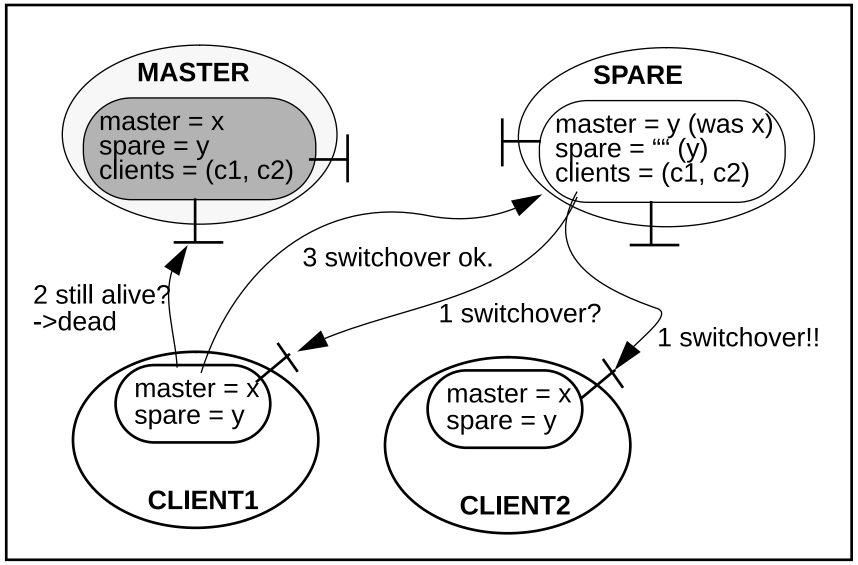


2 On recovery of a spare y, the spare is told of all the clients and then the clients are told of the new spare.
Heartbeats can then be started at the spare. Failure detection at the spare can be started on receival of the 1st heartbeat.

## Spare Switchover contd,



3 The client’s creation of a sparing interface (with the master address) will effect registration of the client’s address to the master which will then relay the address to it’s spare. The master will then send the spare’s address to the client.



4 On master process/ machine failure, heartbeats will fail and the spare shall initiate switchover. An attempt is made to switchover each client to the spare. To determine valid switchover, each client asserts that the master is truly dead before acknowledging switchover ok.
If the master is not dead, then the client shall fail switchover, and the spare can then back down from initiating switchover. On link repair, the master reactivates failure detection on the spare by re-sending heartbeats.

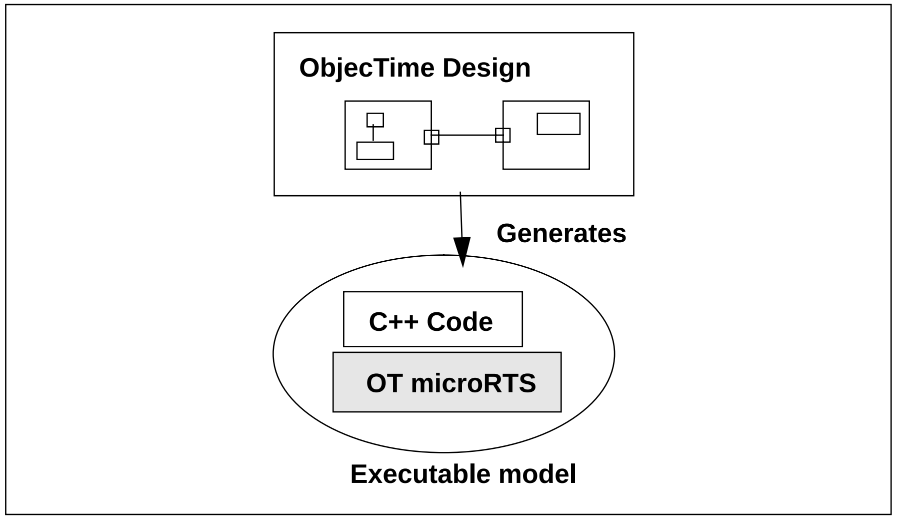
- **In diagram 4, there are several ways to determine master-spare network failure or true master server failure, an example is the pessimistic approach - fail switchover if any client can access master.**

## Sparing Transparency

- Transparency allows for separation of resiliency concerns from specific application functionality;
    - Allows programmers to focus on main system functionality
    - Caters for future resiliency changes without having to change the systems functional design.
- Transparency implications on Clients that use spared services are that they need to mechanisms to perform switchover. Transparency on spared service is easy using a wrapped handle.
- For developing resilient applications, using conventional C++ & IDL would probably employ preprocessing of C++ classes and adherence to abstract base classes.
- However in this particular approach, resiliency transparency has been integrated into a CASE tool.

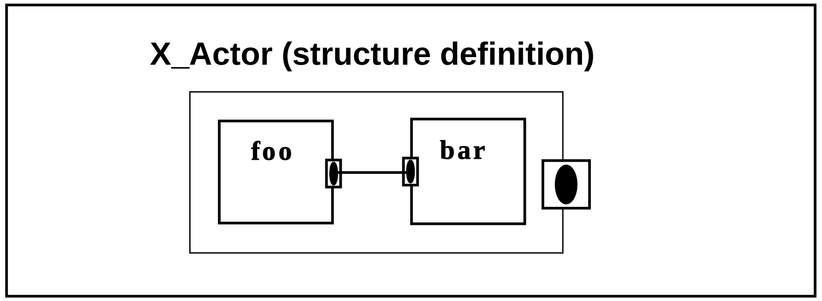
## Introduction into the ObjecTime CASE tool

- ROOM == Real time Object Oriented Modelling methodology originally created to model real time telecommunications systems.
- Originally developed by Bran Selic within BNR
- ObjecTime is the graphical tool developed to support ROOM.
- ObjecTime provides system modelling and design of arbitary complex event driven real time systems and provides for simulation and rapid prototyping.
- Completes the development cycle by supporting the generation of C++ code to run on its external micro run time system - virtual machine to execute models.



## ObjecTime Concepts

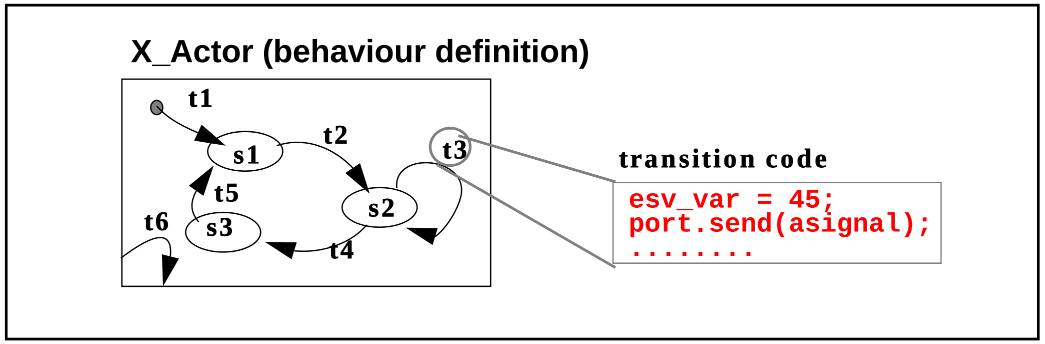
- ROOM provides a *hierarchical decompositional* approach to object oriented design.
- Fundamental building blocks (classes) are called *actors* which possess structural and behavioural properties.



- An actor's structural definition may contain references to nested actor instances.
- Each actor reference has a user specified name.
- Communication *ports* are declared by actors for sending out and receiving messages.
- Two actors may communicate through a *binding* which is basically a connection between two ports.
- Communication can also be done through layered services - *SPP*s and *SAP*s - remove the need for explicit bindings.

## ObjecTime Concepts contd,

- To specify an actor’s behaviour, finite states are defined graphically with transitions from one state to another.

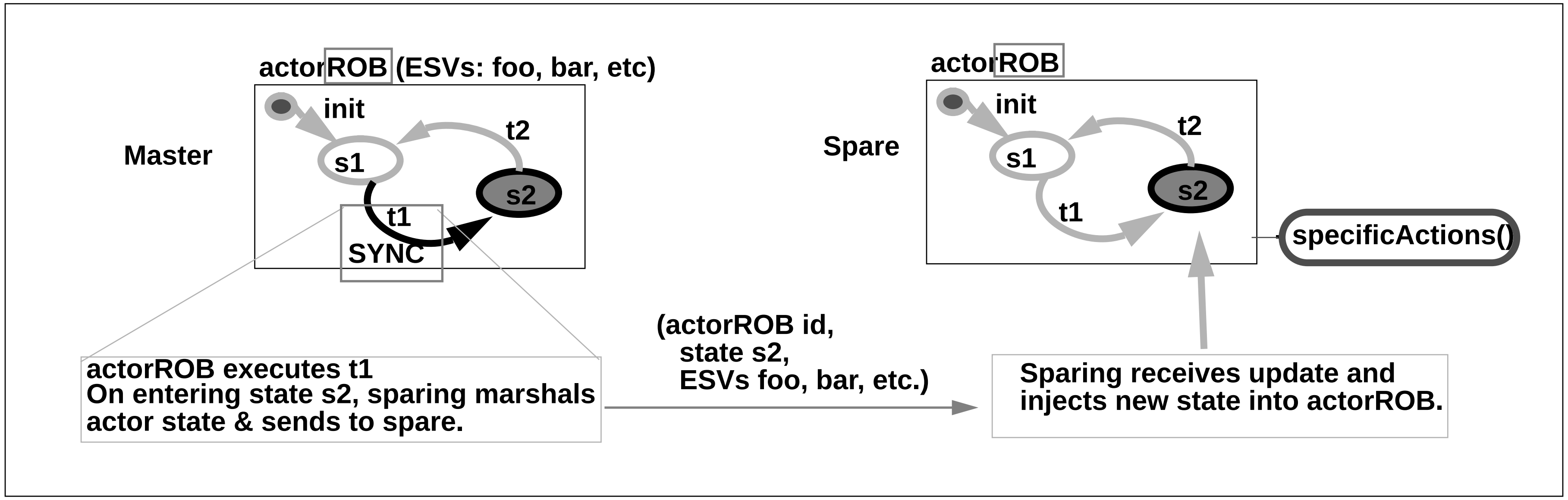


```
esv_var = 45;
port.send(asignal);
........
```

- An actor instance is always in one state.
- An event is triggered when the actor receives messages from other actors through its ports or via timer events.
- Transitions contain code which is run on the occurrence of a particular event that is appropriate to an actor in a particular state.
- An actor’s definition closely resembles a C++ class definition, where member data are termed extended finite state, which can be modified via transition code.

## Sparing Transparency for the ObjecTime CASE tool

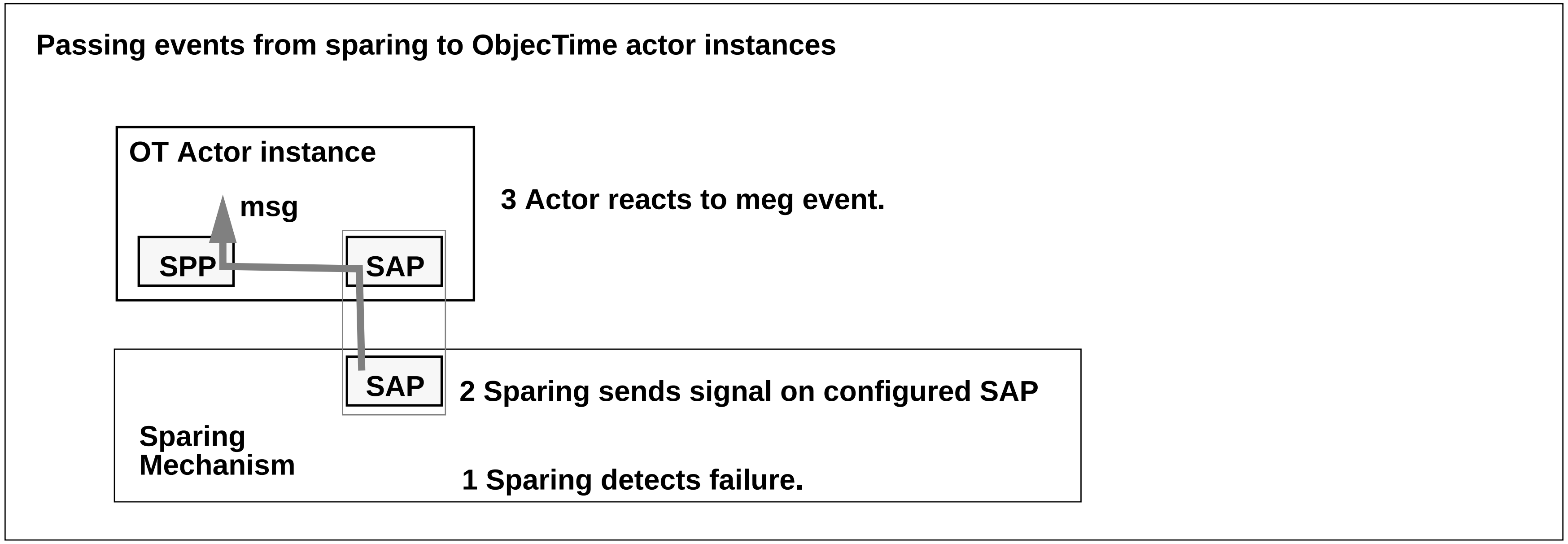
- **Declarative API for Resiliency:**
  - **Declaration of resilient objects at an Actor instance granularity.**
  - **Naming convention used for actor reference names.**
  - **Simply postfix of “ROB” to name of actor reference names.**
  - **Deterministic actors at the spare could run via the “RUNNABLE” postfix.**
  - **Declarative mechanism for system designers to specify synchronisation points.**
  - **Synchronisation points specified in transition code.**


actorROB (ESVs: foo, bar, etc)
Master
init
s1
t2
s2
t1
SYNC
actorROB
Spare
init
s1
t2
s2
t1
specificActions()
(actorROB id, state s2, ESVs foo, bar, etc.)

actorROB executes t1
On entering state s2, sparing marshals actor state & sends to spare.

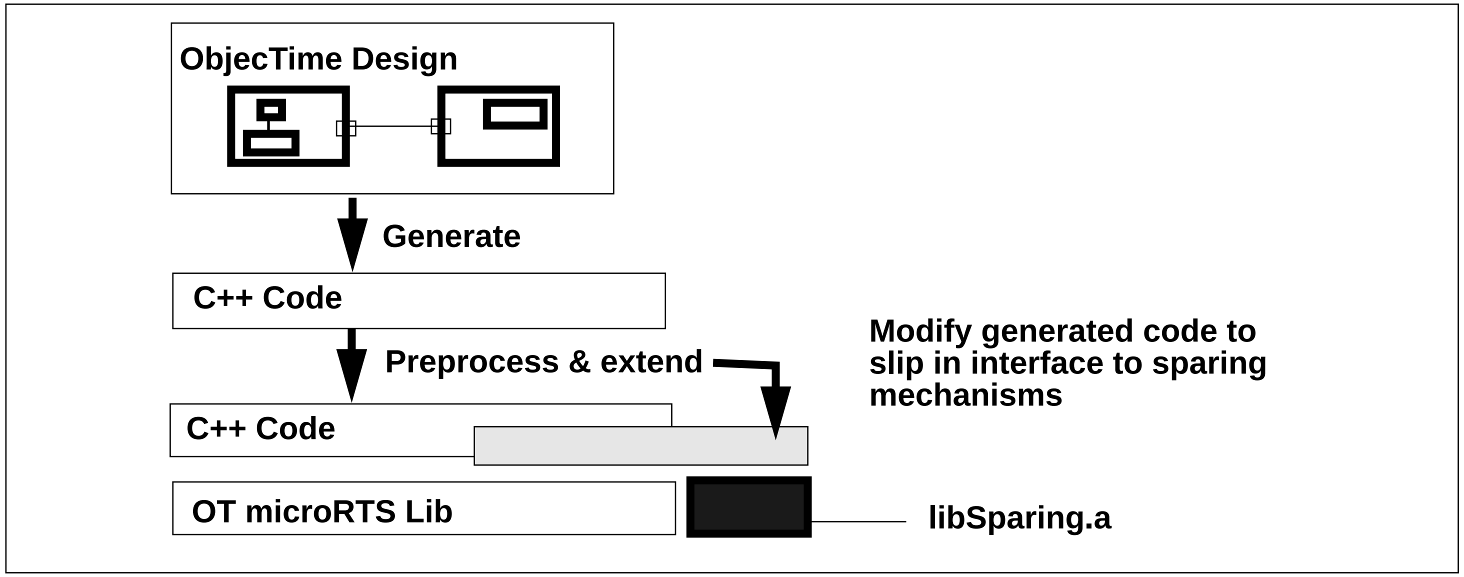
Sparing receives update and injects new state into actorROB.

- **Passing sparing events back into ObjecTime**
  - **Sparing events such as failure detection is relayed back to the ObjecTime model as an ObjecTime signal.**


Passing events from sparing to ObjecTime actor instances
OT Actor instance
msg
SPP
SAP
3 Actor reacts to meg event.
SAP
2 Sparing sends signal on configured SAP
Sparing Mechanism
1 Sparing detects failure.

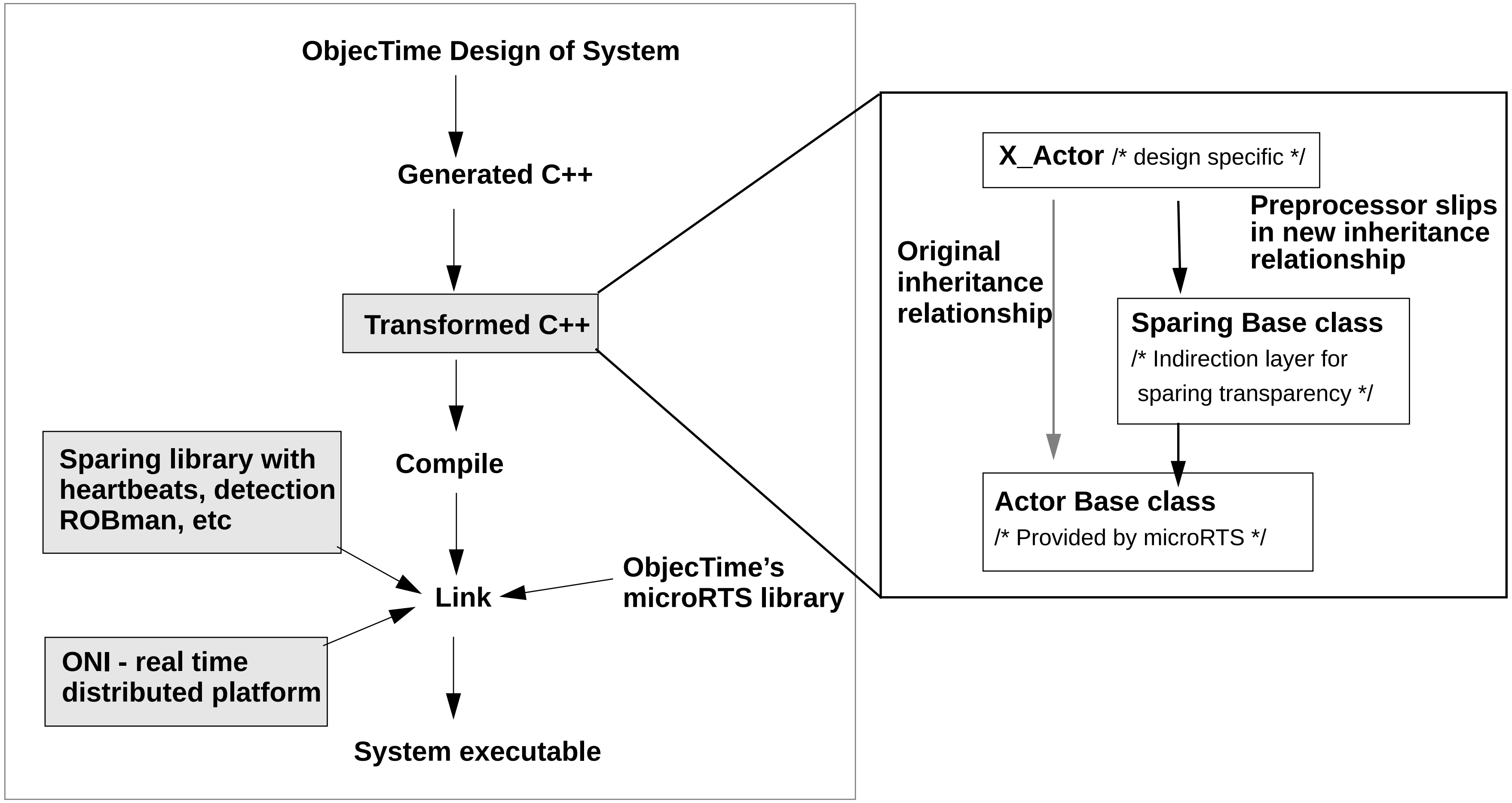
  - **ObjecTime actors explicitly setup SPP and corresponding SAP.**
  - **Actor configures sparing to send it signals corresponding to particular events that it wants to know about.**
  - **On the event, corresponding signals get sent through the configured SAP.**

- **Resiliency integration with the CASE tool:**



- **A preprocessor tool is run after ObjecTime code generation.**
- **Generates marshalling routines for each actor class.**
- **Resiliency functionality added to the ObjecTime model implementation by extending the base hierarchy of the generated C++ code.**
- **From the declarations of reference names, the extended base class identifies each resilient object.**

# Overall building process for spareable ObjecTime implementations


ObjecTime Design of System
Generated C++
Transformed C++
Compile
Sparing library with heartbeats, detection ROBman, etc
Link
ObjecTime's microRTS library
ONI - real time distributed platform
System executable
X_Actor /* design specific */
Original inheritance relationship
Preprocessor slips in new inheritance relationship
Sparing Base class
/* Indirection layer for sparing transparency */
Actor Base class
/* Provided by microRTS */

•

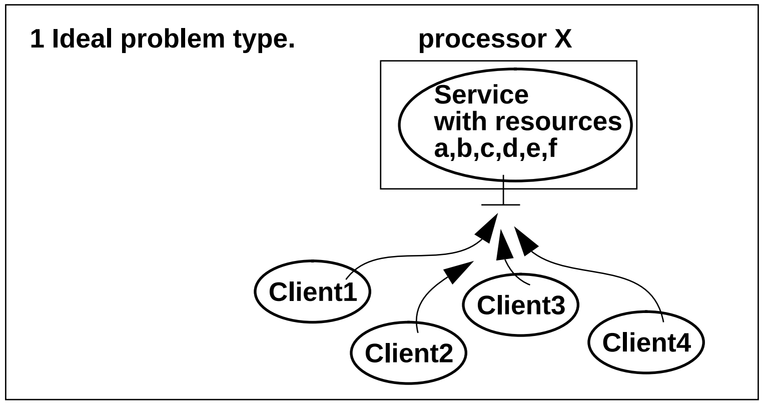
## Sparing and ObjecTime Summary

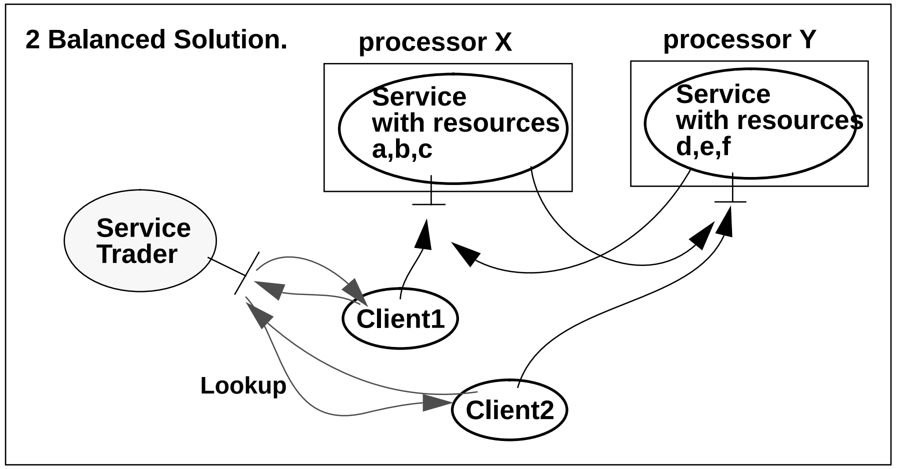
- **ObjecTime enforces the maintainance of mapping model designs to implementations.**
- **Text descriptions can also be attached to individual model components to give a completeness feel for  the development documentation process.**
- **Sparing is a simple approach for high availability that can be applied to any distributed system.**
- **Clients to spared servers only need proxy extensions to handle switchover.**
- **Sparing approach works well with ObjecTime models;**
    - **ROOM enforcement of actor structuring make replication of object context simple.**
    - **Enables spare application objects to remain in context in readiness for switchover.**
- **Simple transparency approach;**
    - **Simple declarative for enabling of preprocessing.**
    - **Preprocessing approach fits in neatly into the ObjecTime code generation phase.**

**Other interests: Sparing & load balancing, ObjecTime and distributed systems.**

## Other uses for Sparing - Load Balancing ?

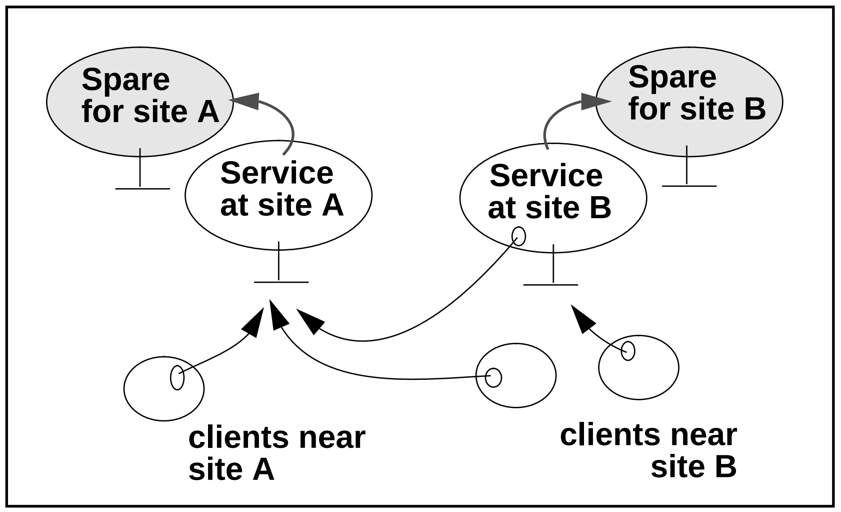
- The aim of load balancing is all about adding more processors to gain more throughput.
- Responsibility of the designer to develop the distributed solution which typically involves *fragmentation* and *replication* of data/resources.
- Load balancing easier where there is a clean resource and work split.

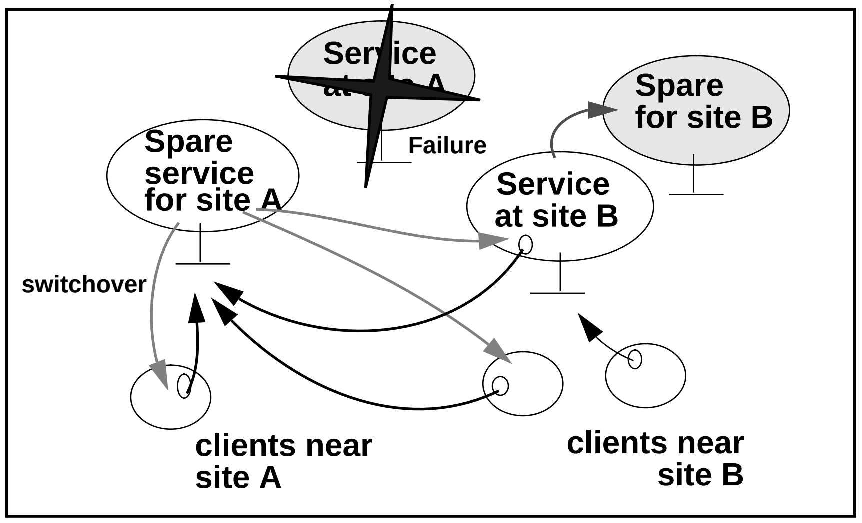




- Complex where replication of services with duplicated resources may result in inconsistency.
- Sparing does not offer a load balancing solution - although a simple one maybe to off load read only operations to the spare.

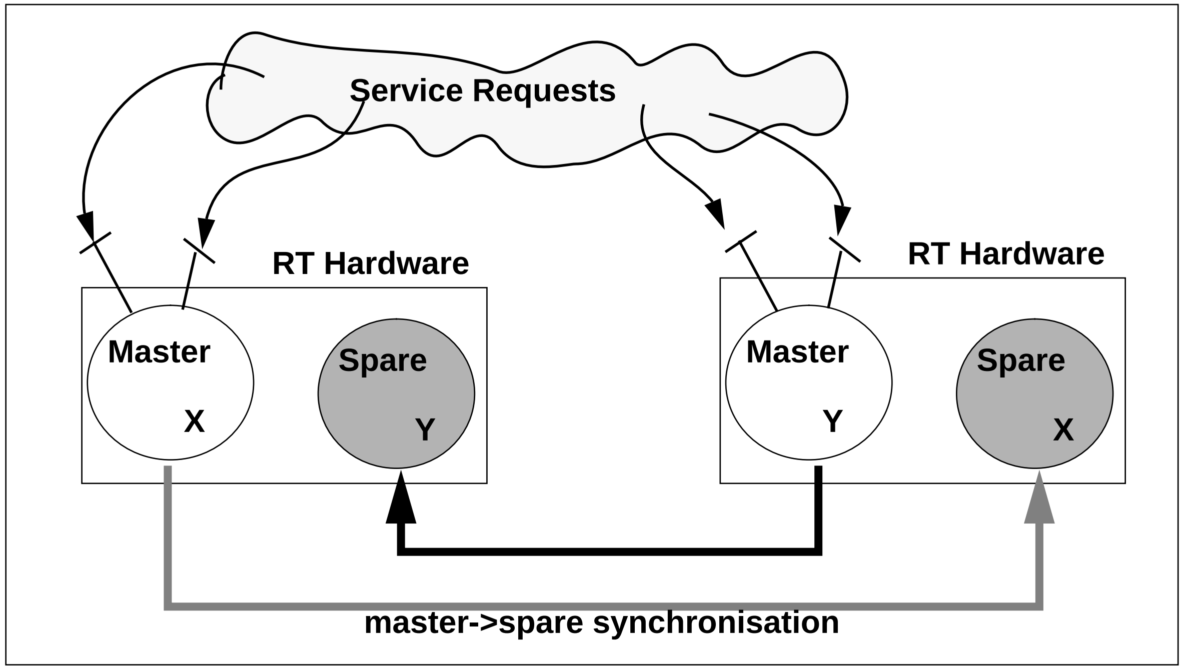
- **Distribution of applications to enable load balancing only offers partial availability in the presence of failures.**
- **Sparing should be *independent of load balancing concerns*.**
- **Sparing should be added to distributed solutions for full high availability.**





- **But can we do better out of sparing to exploit the extra processor?**
- **Possible to exploit the redundant spare in the processing of read-only transactions.**
- **Assuming that work load can be split cleanly, 2 or more independent master and spare processes can be co-located on the same machine**

## Sparing and Load Balancing, contd,

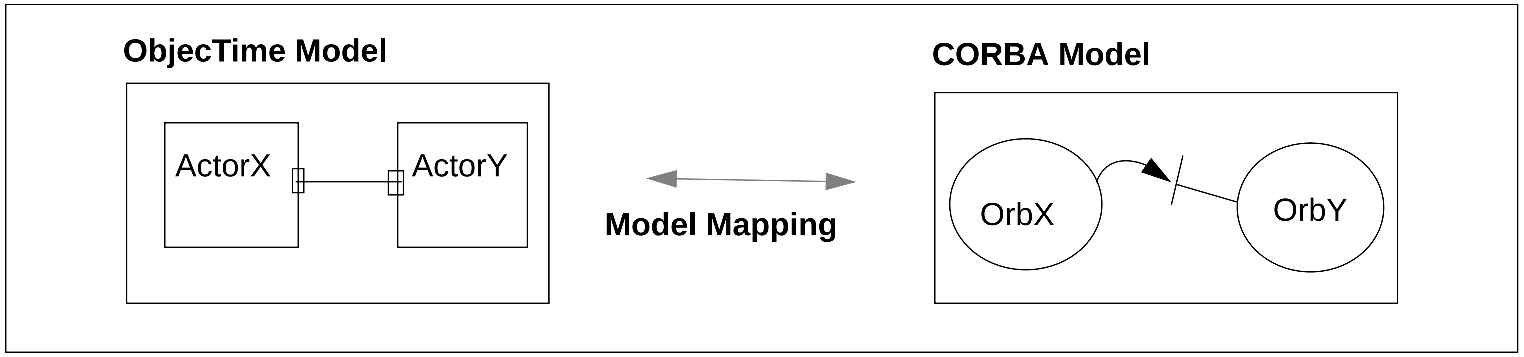


- Ideal when number of synchronisation messages are low.
- Makes full use of both machines if spare is doing relatively little.
- Little increase in overhead for master and spare management.
- One machine may become overloaded when one machine goes down.

## ObjecTime and CORBA

- Current CORBA development approach pros:
    - CORBA - standard for interoperable distributed systems.
    - Can use simple environment for C++ & IDL development.
    - To some degree, provides for rapid prototyping & development through code generation from IDL - compared to writing stubs directly.
- Cons
    - Analysis, modelling and design of servicing objects not enforced or consistent amongst distributed components.
    - No graphical design tool to ensure mapping of design onto implementation - relies on quality procedures, documentation & configuration management tools.
- ObjecTime provides for modelling of distributed systems but does not generate distributable implementations.

- ObjecTime model concepts have similarities with CORBA concepts.



  - Each actor is an autonomous object.
  - ObjecTime protocol definitions map onto interface specifications and actor ports map onto interfaces.
  - ObjecTime messages sent between actors map onto messages sent between ORBs.
- Opens up interesting area for a CASE tool to model and generate CORBA systems.
- A prototype has been developed to show ObjecTime and CORBA interworking.