THALES

OMG RTWS – LwFT Tutorial

Lightweight Fault Tolerance at Work

Paris - April 17th, 2012

Lightweight Fault Tolerance for Distributed RT Systems (LWFT), Version 1.0

- formal/2012-03-02
- Download v1.0 specification from http://www.omg.org/spec/LWFT/1.0/PDF

Agenda

14:05 CORBA Fault Tolerance

Lightweight Fault Tolerance for Distributed Real-Time Systems

Server Replica Management

15:35 Afternoon Refreshments

15:55 Object Group Management

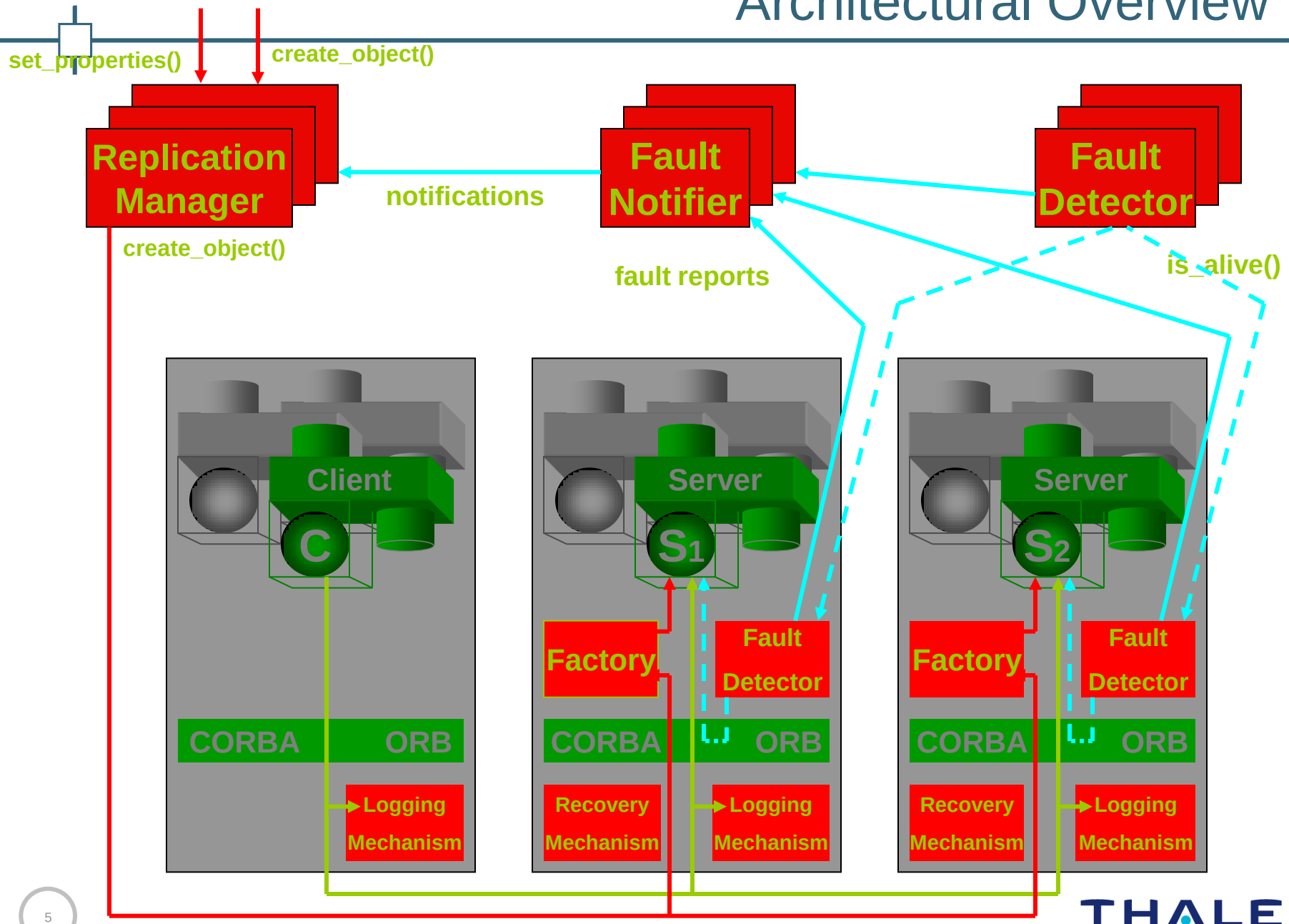
Hello Application

17:25 Conclusion

THALES

- **CORBA Fault Tolerance depends on:**
  - **Entity redundancy (through CORBA Object replication),**
  - **Fault detection, and**
  - **Recovery**

Architectural Overview

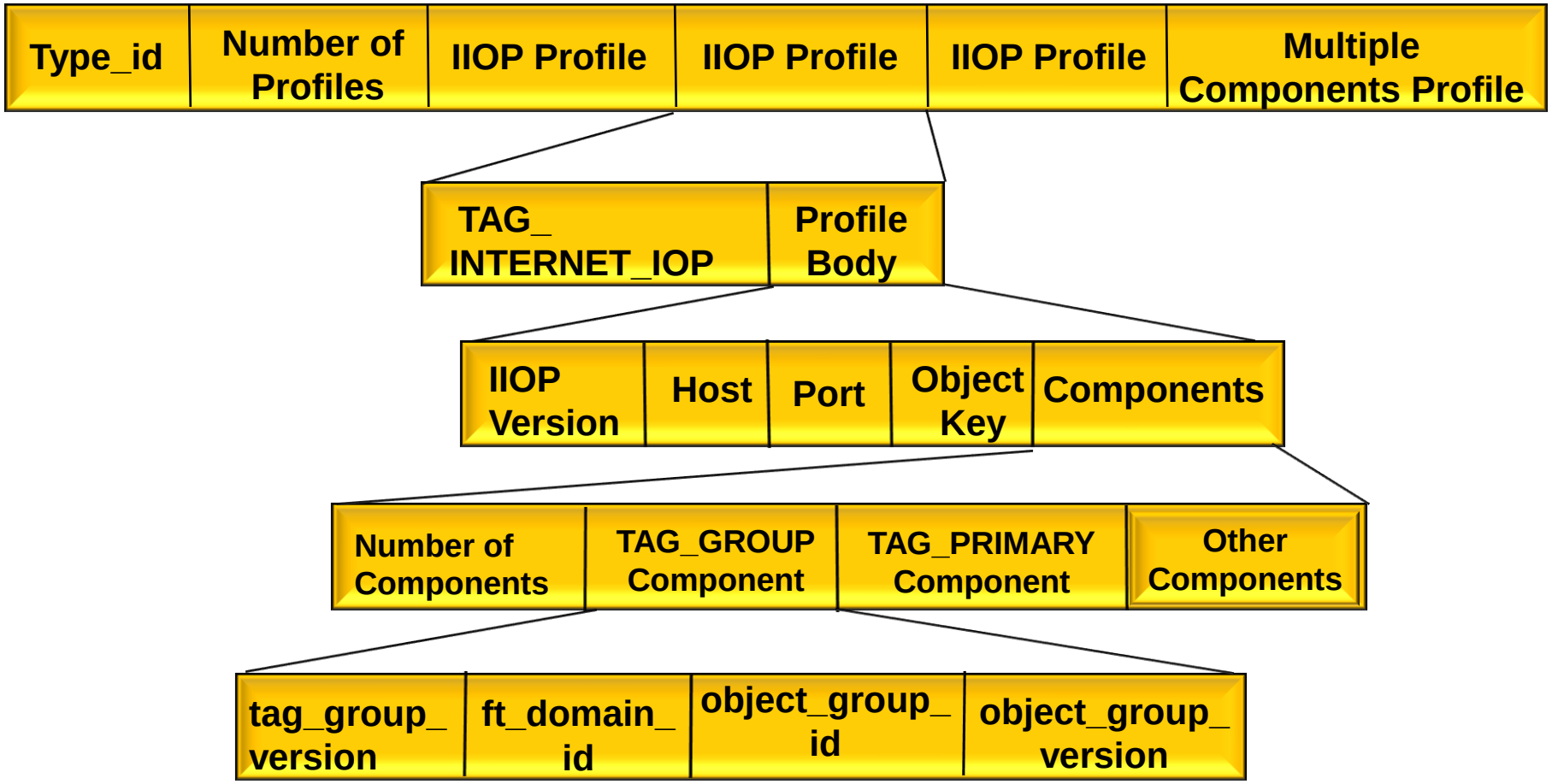


THALES

Object Groups

- Replicas of the object are created and managed as an object group
- Object group abstraction provides
  - Replication transparency
  - Failure transparency
- Interoperable Object Group Reference (IOGR) for the object group as a whole.

- Multiple profile IOR



Replication Style

- Stateless
  - Read-only access to static data
- Cold passive replication
  - Recovery from faults using state information and messages recorded in a message log
  - Slowest recovery from faults
- Warm passive replication
  - Current state of the "primary" replica is transferred periodically to the "backup" replicas
  - More rapid recovery from faults
- Active replication
  - Every replica executes the invoked methods
  - Very rapid fault recovery

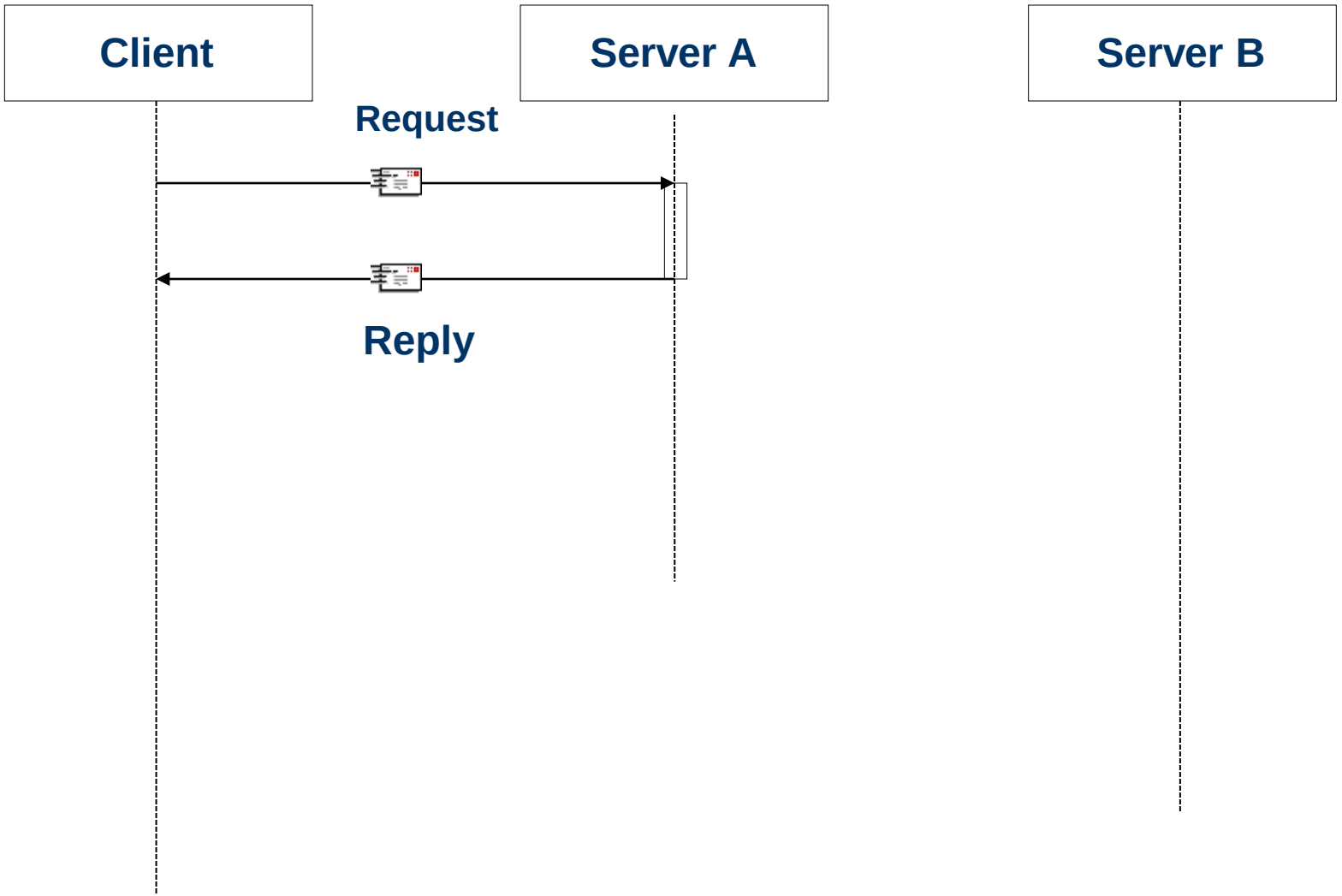
Fault Management

- Fault detection
  - detecting the presence of a fault in the system and generating a fault report.
- Fault notification
  - propagating fault reports to entities that have registered for such notifications.
- Fault analysis/diagnosis
  - analyzing a (potentially large) number of related fault reports and generating condensed or summary reports.
- Fault Containment
  - preventing the propagation of faults from their origin at one point in a system to a point where it can have an effect on the service to the user.

Client Redirection

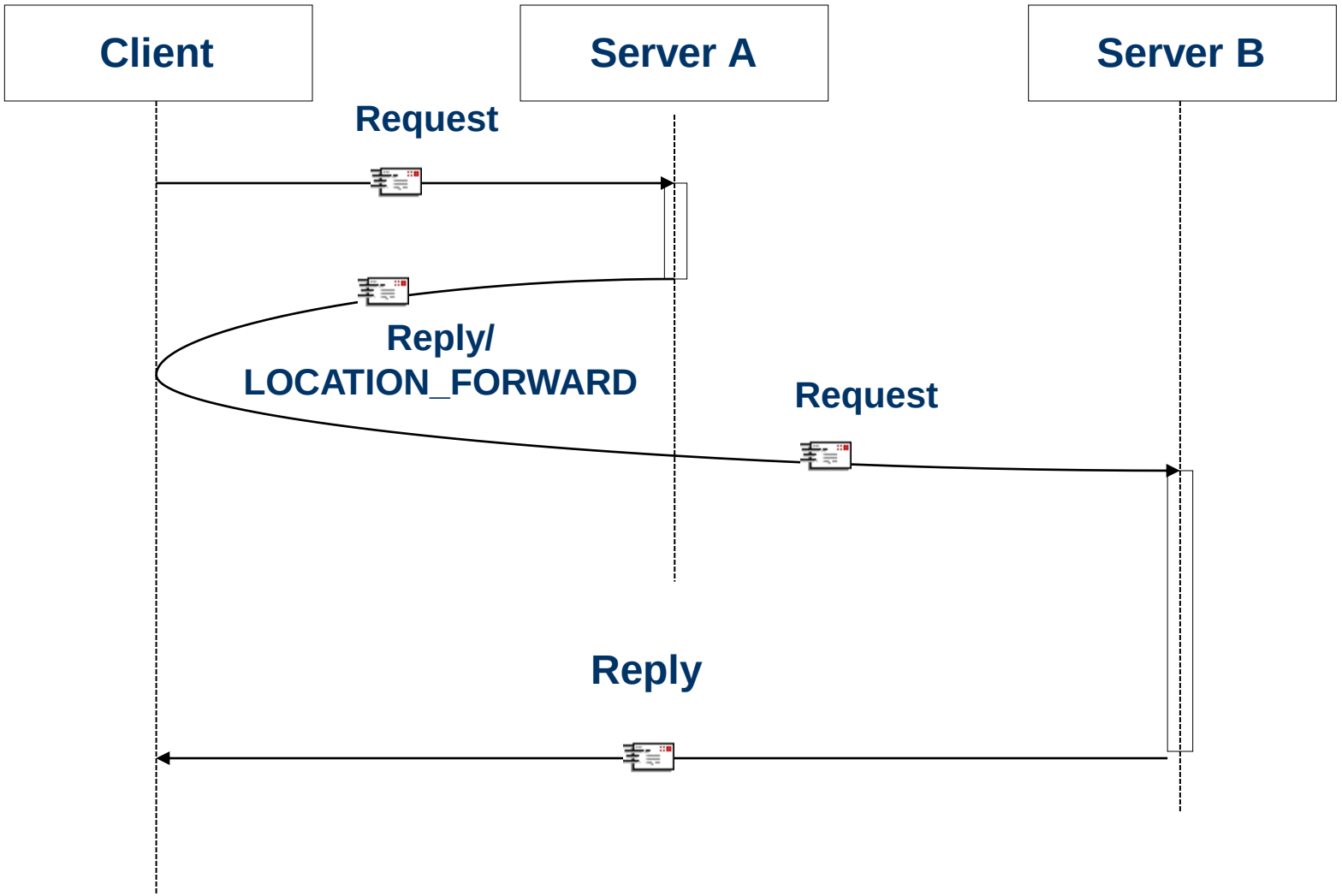
- Failover
  - When a fault occurs in a server replica, client should connect to an alternative server or through an alternative network
  - A client can reinvoke a request, without risk that the operation will be performed more than once
- Lazy provision of most recent object group reference to clients
  - Server is able to determine the version of the object group reference used by the client during the call
- Heartbeating of the server
  - Detect failure of the server or the connection to the server

Remote Method Invocation



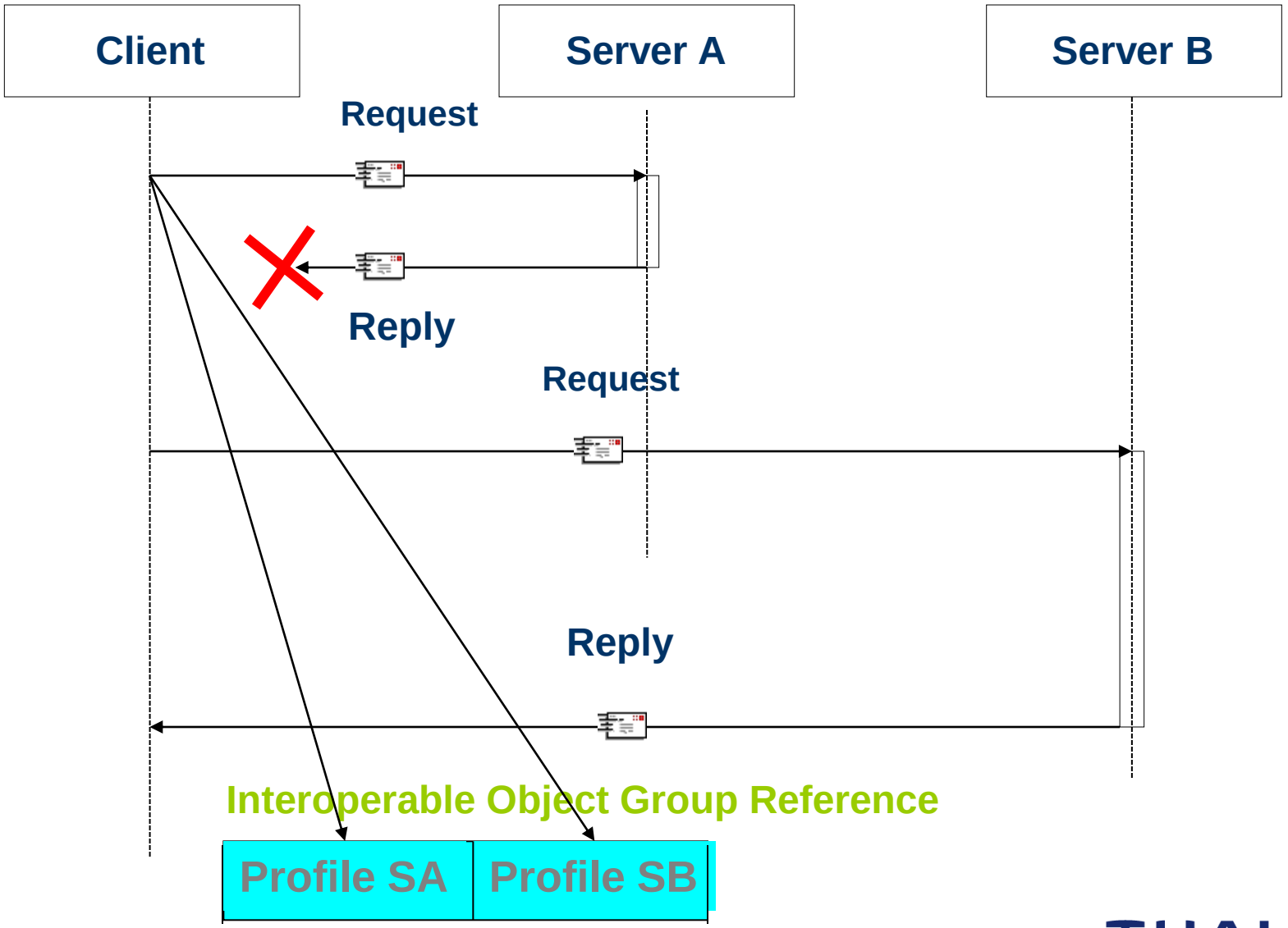
THALES

Transparent Redirection



THALES

Multiple profiles



THALES

FT CORBA Permitted Failover Conditions

	Completion Status	CORBA Exception
Without Transparent Reinvocation	**COMPLETED_NO**	**COMM_FAILURE** **TRANSIENT** **NO_RESPONSE** **OBJ_ADAPTER**
With Transparent Reinvocation	**COMPLETED_NO** **COMPLETED_MAYBE**	**COMM_FAILURE** **TRANSIENT** **NO_RESPONSE**

Must still guarantee

at most once semantics!

THALES

- **How to ensure that a request is not executed more than once under fault conditions?**
- **Detect transparent reinvocation using FT_REQUEST service context and supply previously generated reply.**
  - **FT_REQUEST Service Context**
    - **Client ID**
    - **Retention ID**
    - **Expiration Time**

FT CORBA Limitations

- Fault-tolerance provides solutions to mitigate reliability problems:
  - Fault-tolerance requires some kind of redundancy, fault detection and recovery
- Current FT CORBA specification is not always fully applicable:
  - Users often have their own mechanisms for persistency, state transfer … etc. (DB, DDS, PSS, Files, legacy …)
  - There are many kinds of fault detectors
  - Object granularity is too fine for fault detection
  - Support for RT Fault tolerance implies:
    - Predictable Recovery time
    - Predictable Reference update/refresh

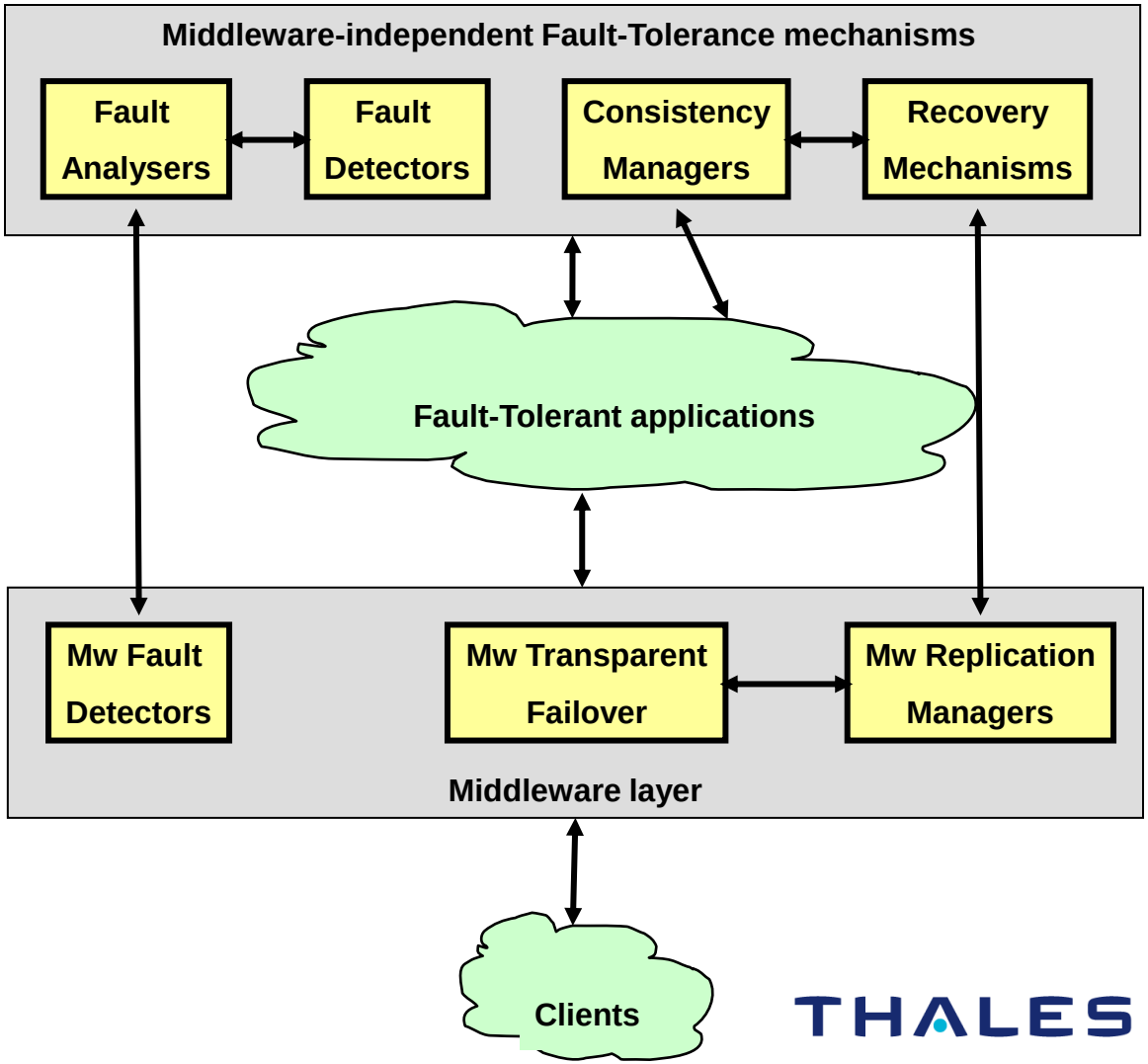
THALES

Lightweight Fault Tolerance for Distributed Real-Time Systems

Over-all design rationale (RFP)

Micro-kernel architecture applied to fault tolerant systems

- Middleware independent
  - consistency management
  - fault analysis, containment and recovery
- External fault tolerance mechanisms for
  - fault detection
  - replication management
  - transparent failover.
- Support for both active and passive replications



THALES

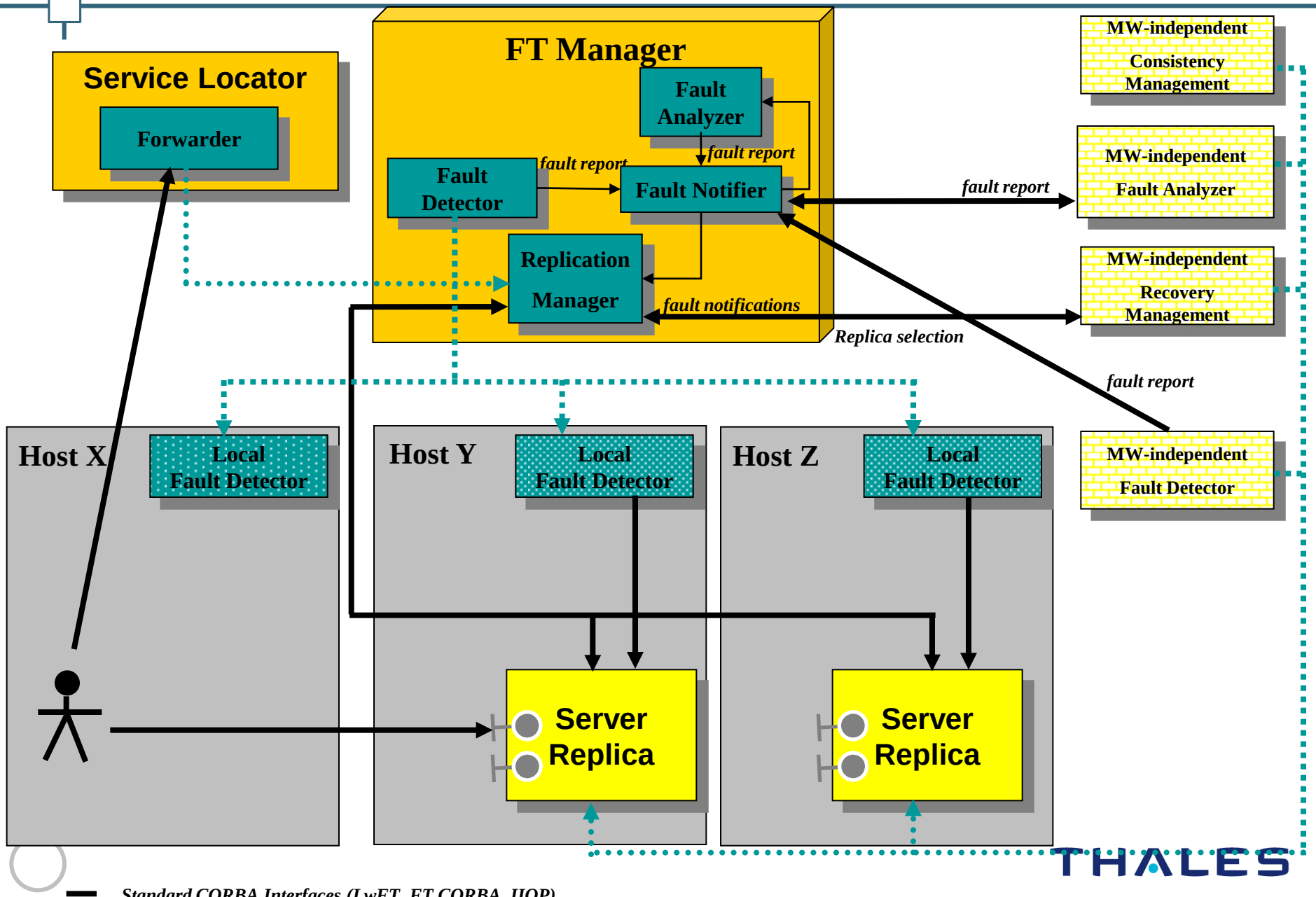
FT-enabled middleware

- FT-enabled middleware is responsible for providing the group abstraction needed for a reliable architecture through middleware Replication Management and transparent failover.
- The FT middleware may also rely on some middleware fault detectors for monitoring the middleware infrastructure.

Middleware-independent FT infrastructure

- The FT-enabled middleware relies on external mechanisms for fault detection, fault analysis, and consistency and recovery management.

General Architecture



Lightweight Fault Tolerance Highlights

Server Replication Management Independent from access to service

- No assumption on the way application offer/use services (Client/Server, Data Publish/Subscribe … etc.).

State consistency managed by applications

- Application state consistency decided and managed by application-supplied mechanisms.

Naming Object Groups

- To ease application bootstrapping and service localization (Location/Forwarding) object groups are named using human readable schemes.

Lightweight Fault Tolerance Highlights

Explicit control of CORBA requests failover

- Application developers can control how and if failover occurs thanks to ClientFailoverSemanticsPolicy policy.
- FTCORBA and CORBA Messaging invocation timeouts still apply (RelativeRoundtripPolicy, RequestDurationPolicy … etc.).

Fault Tolerance Current

- Applications free to enforce the at most once semantics of CORBA invocations thanks to the FTCurrent object.
- Full access to invocation identification to detect repetitive request invocations.

Location Forwarding

- Use of object groups when failure and replication transparencies are required based on standard forwarding mechanisms on either standard IORs or Interoperable Object Group references (IOGRs).
- A Service Locator architectural component is responsible for locating/retrieving latest group object reference when all previously known profiles have failed.

Server Replica Management

- Support for both cold-passive and warm passive replication styles regardless of the interaction style (C/S, Pub/Sub).

Basic Object Group Management

- Server Replica Management with support for client/server CORBA interactions based on standard IORs.

À la Implementation Repository

Extended Object Group Management

- Basic Object Group Management with support for PortableGroups, FTCORBA IOGR's for enforcing at most once semantics, and support for Fault Detection and Notification.

Changes to existing specifications

LwFT specification adds the following to CORBA 3.1 specification:

- A new policy object: ClientFailoverSemanticsPolicy
- Two new POA Policies: ObjectGroupMembershipPolicy, and ObjectGroupNameResolutionPolicy
- One Current object : FTCurrent
- Five new ObjectIds for ORB::resove_initial_references:
  - “FTCurrent”, “FTServiceLocator”, “FTObjectGroupManager”, “FTServerGroupManager”, and “FTServerManager”.
- ORB Configuration parameters:
  - -ORBKeepAlive, and –ORBFTLocation
- One pseudo operation: FT_Locate.

LwFT specification adds the following to FT CORBA specification:

- Add support for location agents in Interoperable Object Group References
- A new service context: FT_GROUP_ID

THALES

Server Replica Management

Server Groups vs. Object Groups

Object groups provide access to services offered by the process group

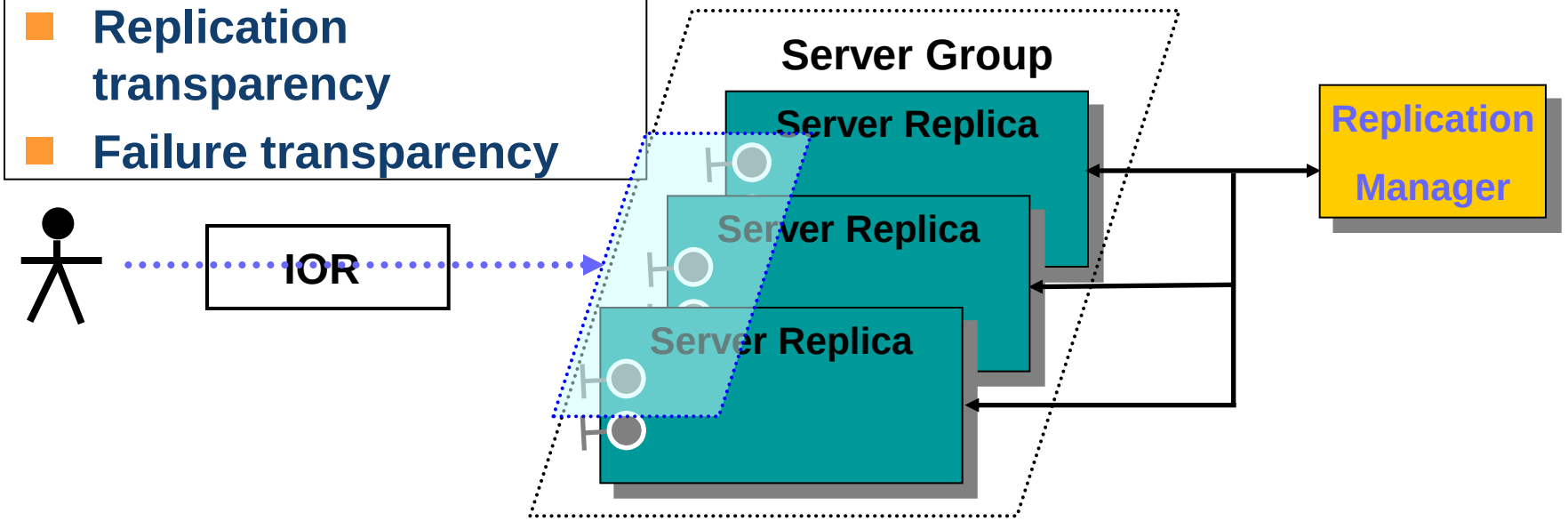
- Well defined failover semantics

Object group abstraction provides

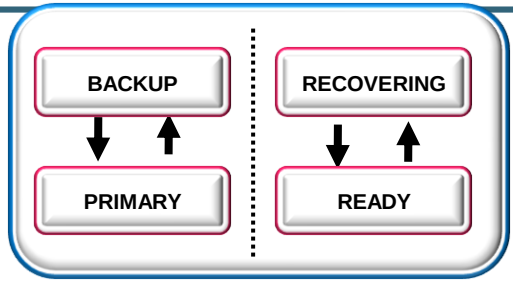
- Replication transparency
- Failure transparency

Server Group abstraction for Replication Management

- [POSIX] Process is the unit of replication for consistency management, failure detection … etc.



THALES



Server Replica Readiness Status

```
typedef long ServerStatusValue;
const ServerStatusValue SRV_STATUS_UNKNOWN    = 0;
const ServerStatusValue SRV_STATUS_RECOVERING = 1;
const ServerStatusValue SRV_STATUS_READY      = 2;
```

Server Replica Replication Role

```
typedef long ServerReplicationRoleValue;
const ServerReplicationRoleValue SRV_REPL_UNKNOWN     = 0;
const ServerReplicationRoleValue SRV_REPL_PRIMARY     = 1;
const ServerReplicationRoleValue SRV_REPL_BACKUP      = 2;
const ServerReplicationRoleValue SRV_REPL_ACTIVE      = 3;
```

Standard server group management via ServerGroupManager interface

→ ORB::resolve_initial_references (*"FTServerGroupManager"*).

***Simple local interfaces for applications*:**

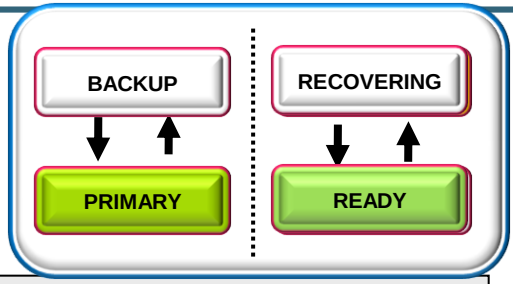
- **Group membership,**
- **Server readiness**

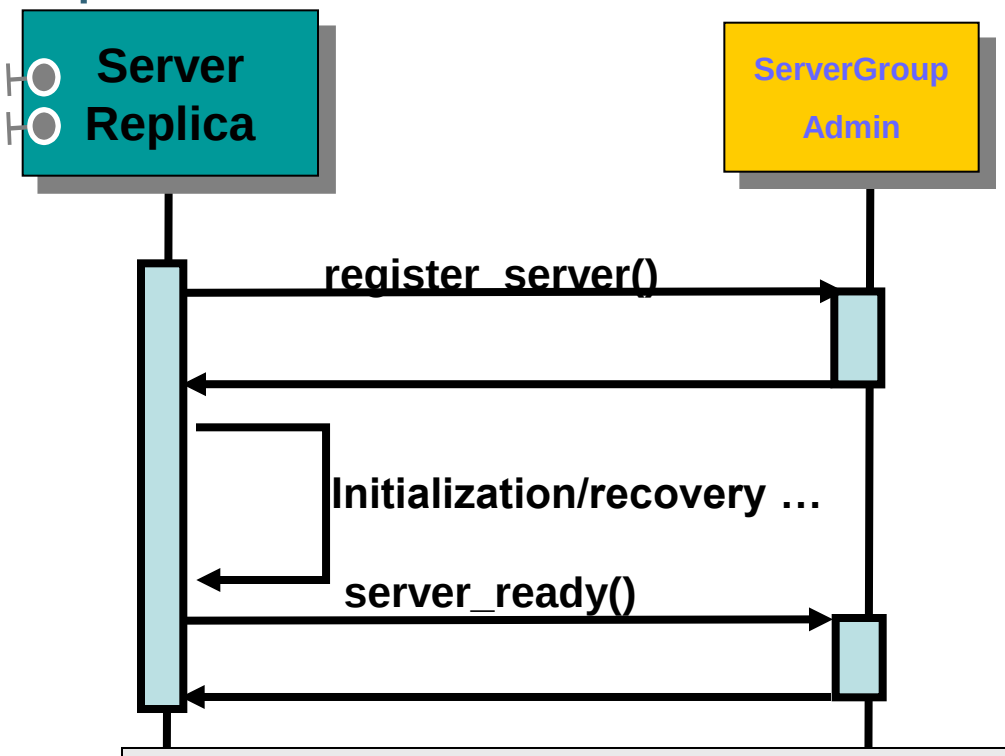
→ ORB::resolve_initial_references (*"FTServerManager"*).

```
interface ServerGroupManager {
    void register_server  (
        inout Location    the_location,
        in ServerID       server_id,
        in Properties     props,
        in ServerCallback callback)
    raises (ServerGroupNotFound,
    AlreadyRegistered, UnsupportedCallback );
    ...
    void server_ready (
        in Location the_location)
    raises (ServerNotFound);

};
```

```
local interface ServerManager {
    attribute Location the_location;
    void register_server ();
    void server_ready ();
    void unregister_server ();
};
```

Example: Start up of a Primary Server





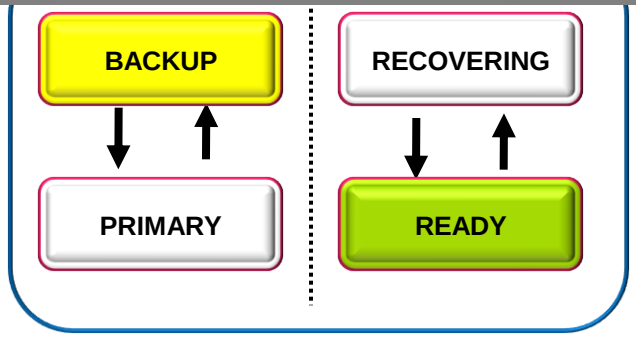

```
local interface ServerManager {
    attribute Location the_location;
    void register_server ();
    void server_ready ();
    void unregister_server ();

};
```

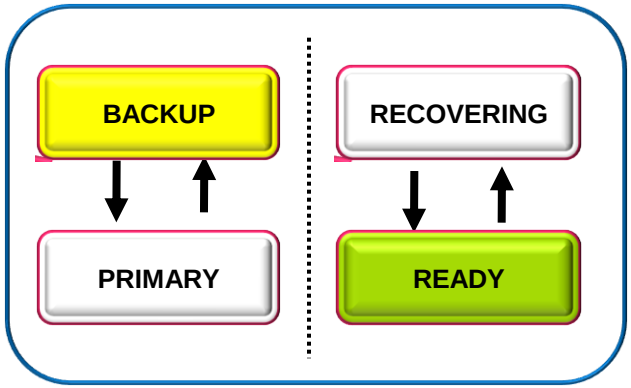
```
CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);

CORBA::Object_var obj = orb->resolve_intial_references("FTServerManager");
LWFT::ServerManager_var sm = LWFT::ServerManager::_narrow(obj.in());

// Join the process group now
sm->register_server ();

// Perform Primary Initialisation or Backup Insertion

// Server replica is ready now.
sm->server_ready ();
```

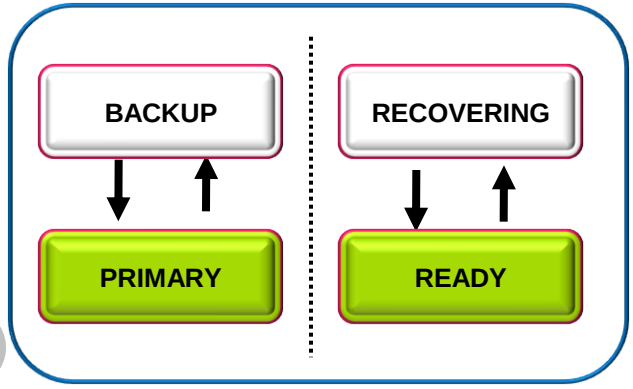
-ORBServerId identifies a server group
-ORBFTLocation identifies the server replica

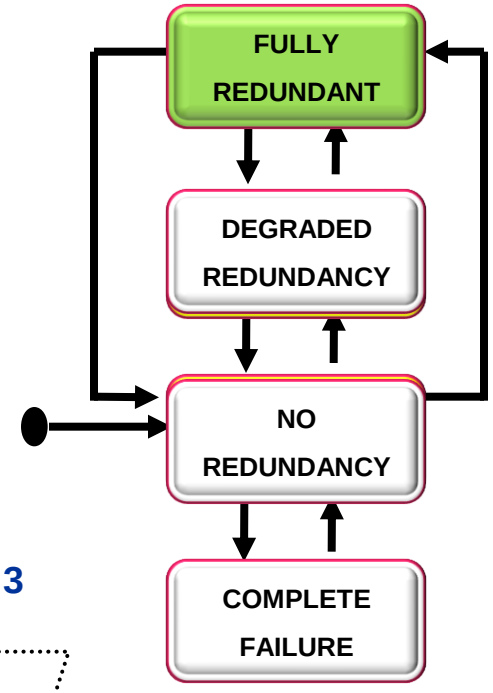
Server Groups vs. Server Replicas

Independent of kind of interaction between service provider and user (C/S, DDS …)!

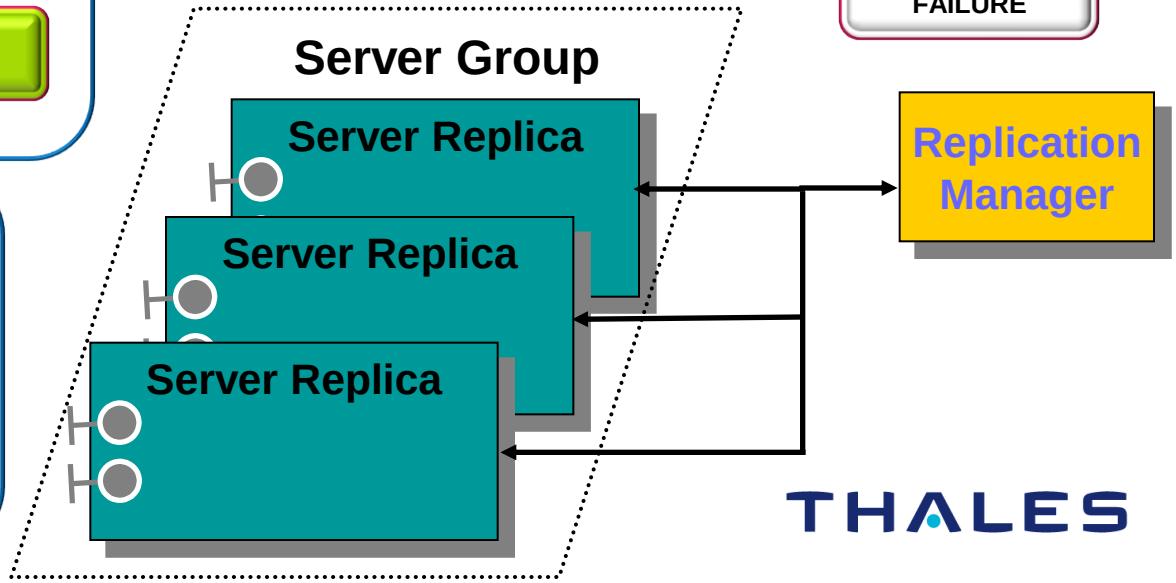








Minimum nb replicas = 3



THALES

RecoveryObserver *interface*

Notified whenever recovery is in progress, .i.e., a backup is being promoted to become a primary.

```
local interface ServerRecoveryManager :
    ServerManager
{
boolean register_recovery_observer
   (in RecoveryObserver rec);
boolean unregister_recovery_observer
   (in RecoveryObserver  rec);
readonly attribute ServerStatusValue
   replica_status;
readonly attribute
ServerReplicationRoleValue replica_role;
};
```

```
local interface RecoveryObserver
{
    void on_recovery();
};
```

Replication Management

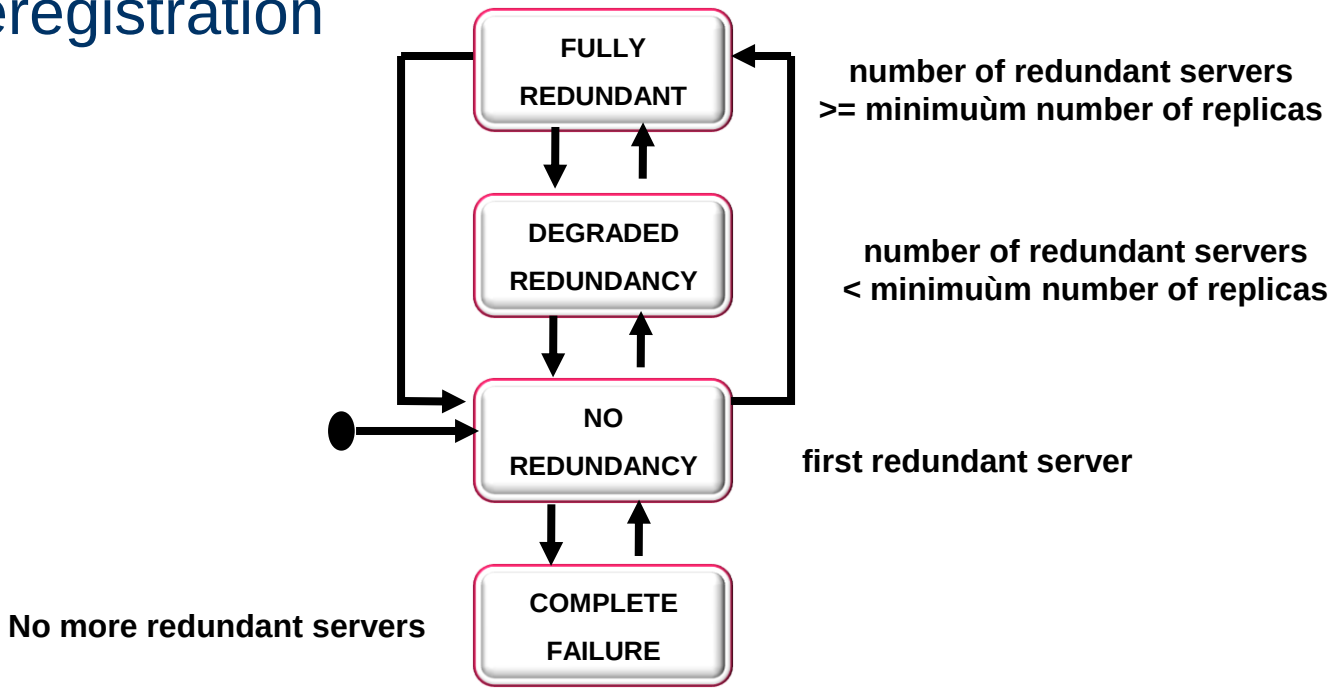
```
CORBA::Object_var obj = orb->resolve_intial_references("FTServerManager");
LWFT::ServerManager_var sm = LWFT::ServerManager::_narrow(obj.in());
LWFT::ServerRecoveryManager_var srm = LWFT::ServerRecoveryManager::_narrow(sm.in());

// Register RecoveryObserver
MyRecoveryObserver * mRec = new MyRecoveryObserver();
srm->register_recovery_observer(mRec);

// Join the process group now
srm->register_server ();
// Perform Primary Initialisation or Backup Insertion
LWFT::ServerReplicationRoleValue role = srm->replica_role();
if (role == LWFT::SRV_REPL_PRIMARY) {
   // Perform Primary Initialisation
} else if (role == LWFT::SRV_REPL_BACKUP) {
   // Perform Backup Insertion
} else {
   // Error!
}

// Server replica is ready now.
srm->server_ready ();

orb->run();

srm->unregister_recovery_observer(mRec);
srm->unregister_server ();
```

LWFT specification has no provision for a service to determine primary location.
Applications should rely on their own means and/or supervision services.

THALES

Advanced Group Membership Operations

Support for notifications following Server replica registration / deregistration



Advanced Group Membership Operations

```
struct ServerGroupMember
{
    Location        server_location;
    ServerCallback  server_callback;
    Properties      props;
};
typedef sequence<ServerGroupMember> ServerGroupMembers;

interface GroupUpdateObserver : ServerCallback
{
  void on_update_group(in ServerGroupMembers members);
};
```

```
interface ServerGroupManagerExt :
ServerGroupManager
{
   void register_observer
        (in ServerIdSeq server_groups,
         in ServerGroupObserver obs);
   void unregister_observer
        (in ServerIdSeq server_groups);
};
```

```
interface ServerGroupObserver
{
  void on_register_server
     (in Location       the_location,
      in ServerId       the_server_id,
      in Properties     props,
      in ServerCallback callback);

  void on_unregister_server
     (in Location the_location);
  void on_server_ready
     (in Location the_location);
};
```

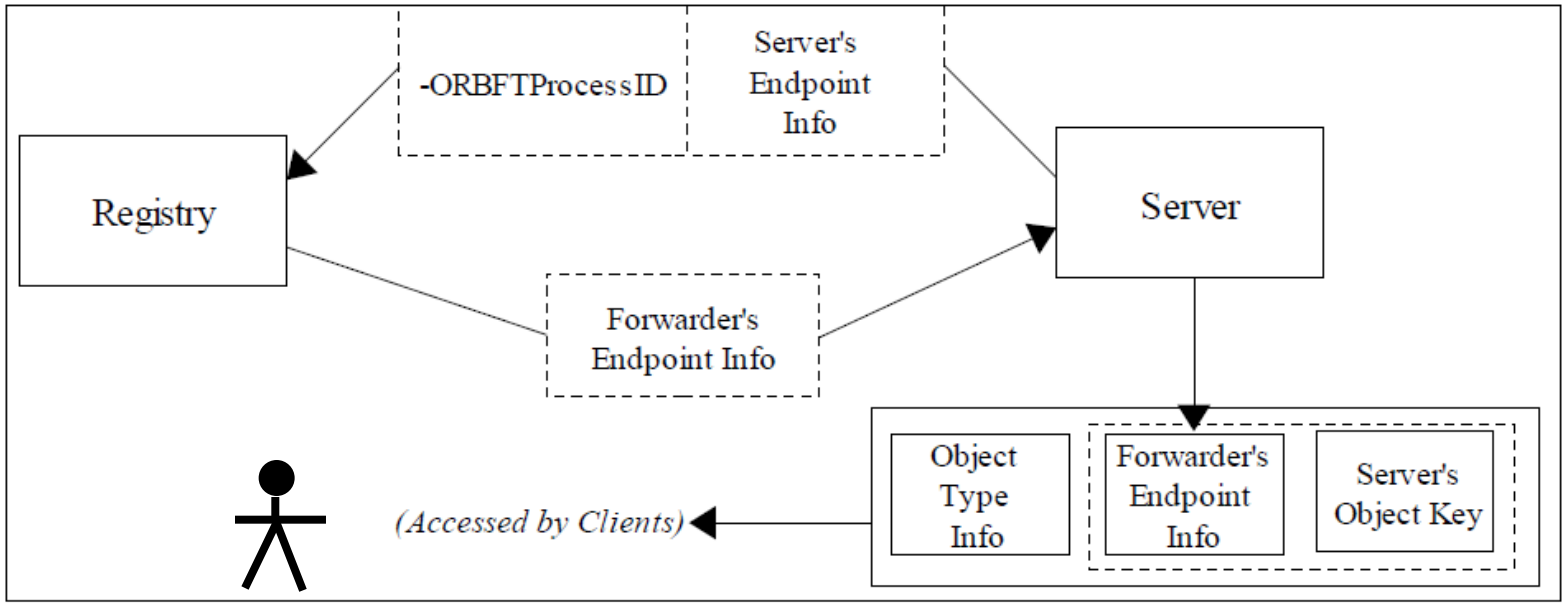
THALES

Object Group Management

Object References (1/3)

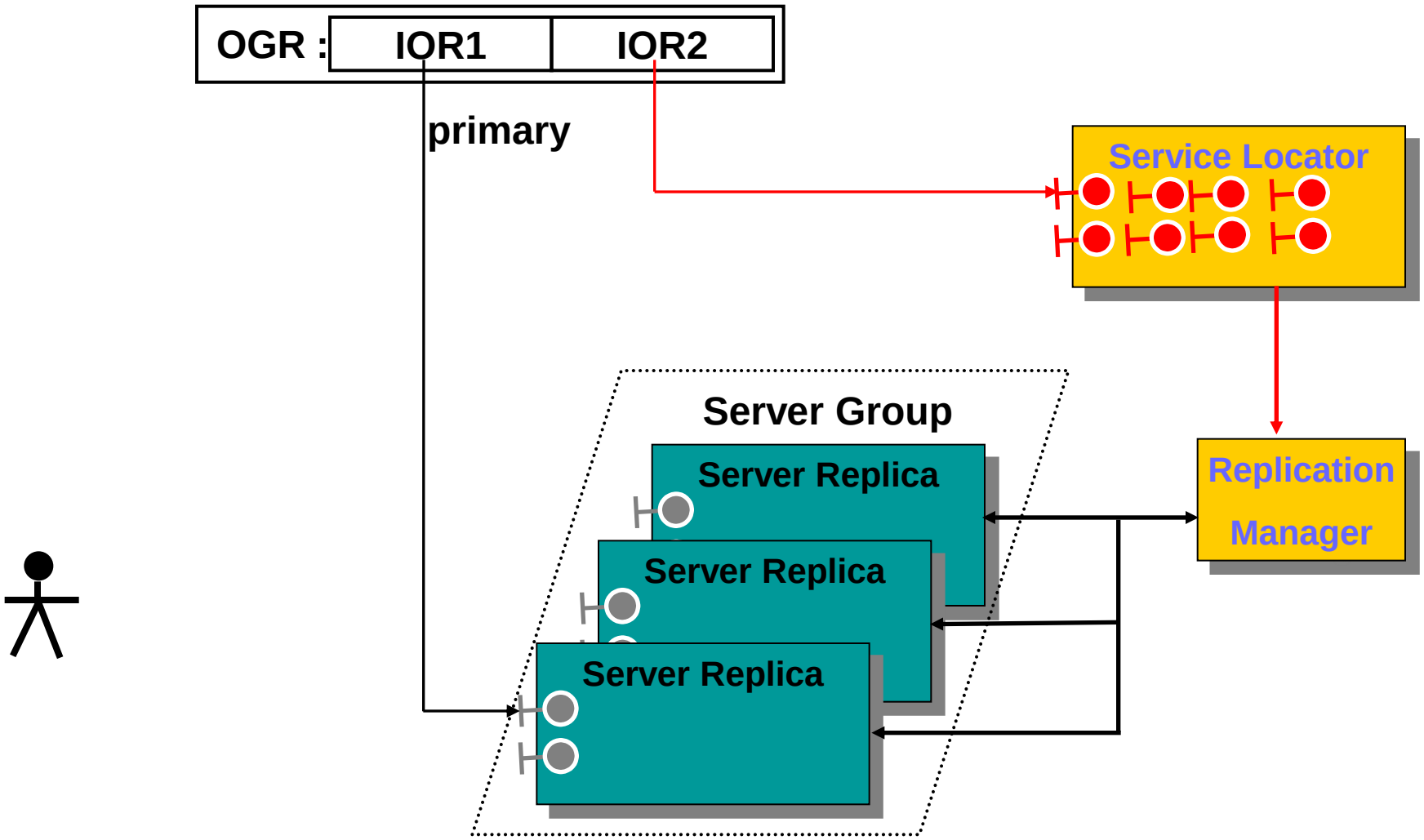
Normal CORBA IOR with transport information replaced with Forwarder endpoint info

- *Implementation Repository approach*



Object References (2/3)

Group reference as either a standard IOR

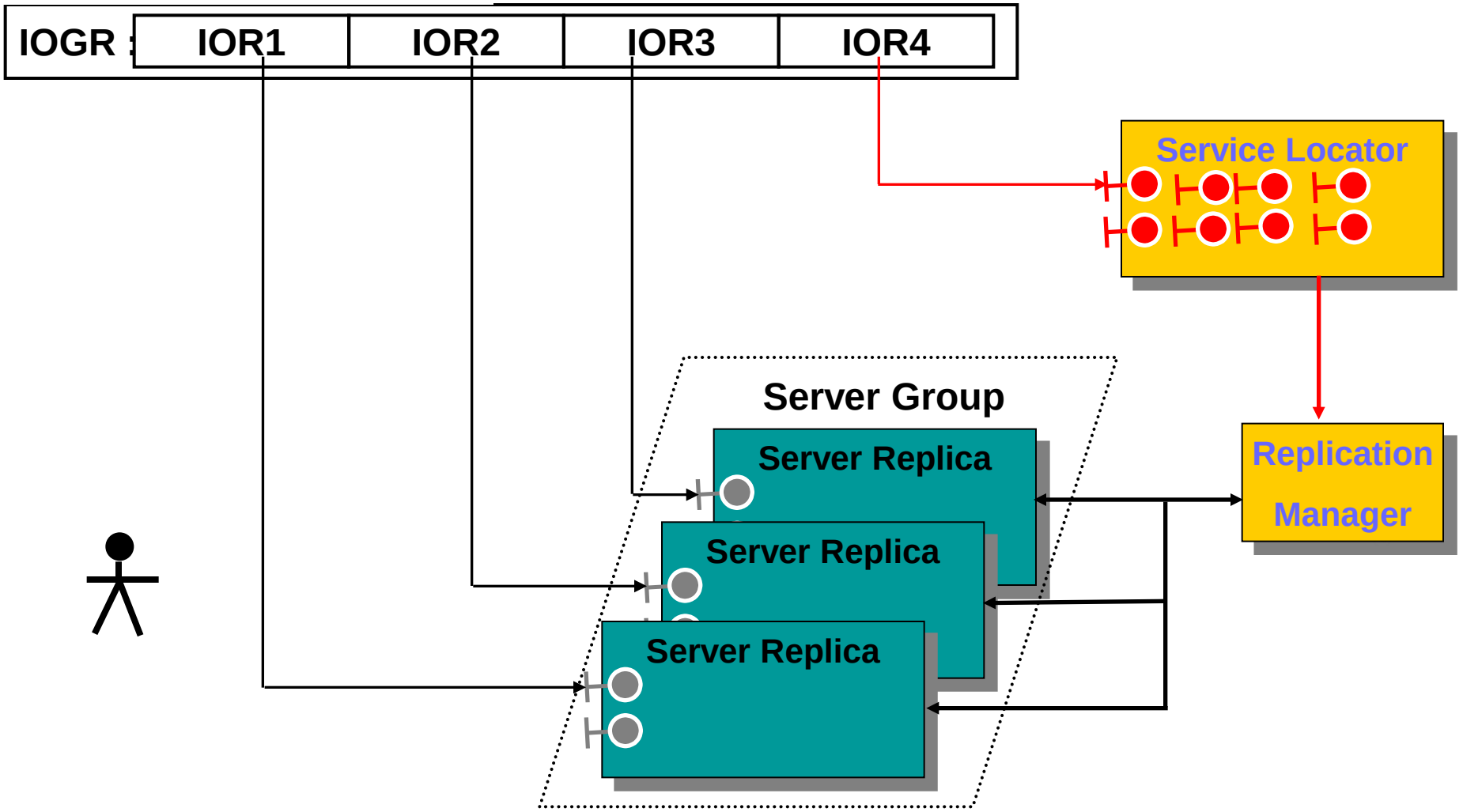


Forwarder as an ultimate fallback!

THALES

Object References (3/3)

Or as an IOGR



Forwarder as an ultimate fallback!

THALES

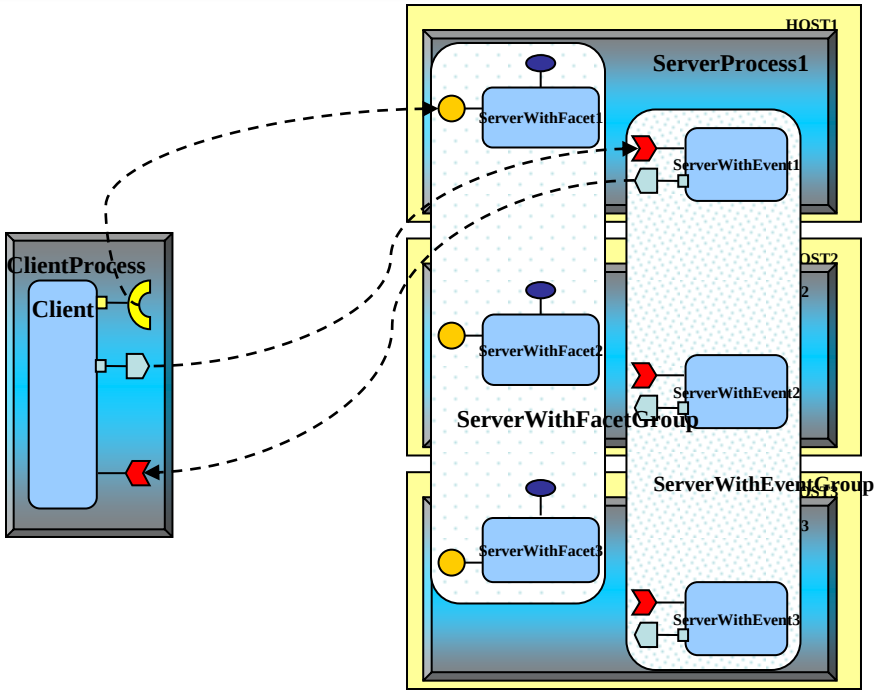
More Policies!

This is lightweight but add 3 new policy objects!

- **ClientFailoverSemanticsPolicy,**
- **ObjectGroupMembershipPolicy, and**
- **ObjectGroupNameResolutionPolicy**

Only needed for Advanced users

CCM Containers hide this additional complexity



THALES

ClientFailoverSemantics Policy

- *ClientFailoverSemanticsPolicy*
  - Explicit control on level of support for reinvocation following failover conditions.
  - Effective in the context of any object reference.

```
enum ClientFailoverSemanticsPolicyValue {
      ORB_DEFAULT,
      BEST_EFFORT,
      AT_LEAST_ONCE,
      AT_MOST_ONCE
};
local interface ClientFailoverSemanticsPolicy :
     CORBA::Policy {
readonly attribute ClientFailverSemanticsPolicyValue
value;
};
```

4 possible values

- **ORB_DEFAULT**
  The ORB behaviour is unmodified from its default.
- **BEST_EFFORT**
  Guaranteed 'best effort' delivery attempt without transparent reinvocation on COMPLETED_MAYBE.
- **AT_LEAST_ONCE**
  Transparent reinvocation on COMPLETED_MAYBE. No mandatory transmission of FTRequestServiceContext.
- **AT_MOST_ONCE**
  As AT_LEAST_ONCE but the client ORB must transmit a FTRequestServiceContext with the request.

Table 23-1 Permitted Failover Conditions without and with Transparent Reinvocation

	Completion Status	CORBA Exception
Without Transparent Reinvocation	COMPLETED_NO	COMM_FAILURE TRANSIENT NO_RESPONSE OBJ_ADAPTER
With Transparent Reinvocation	COMPLETED_NO COMPLETED_MAYBE	COMM_FAILURE TRANSIENT NO_RESPONSE OBJ_ADAPTER

```
module FT {
struct FTRequestServiceContext {
   string client_id;
   long retention_id;
   TimeBase::TimeT expiration_time;
};
};
```

THALES

At Most Once ClientFailoverSemantics Policy

```
CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);

// Create the ClientFailoverSemanticsPolicy
CORBA::Any failover;
failover <<= LWFT::AT_MOST_ONCE;

CORBA::PolicyList policy_list;
// Set the policy
policy_list.length(1);
policy_list[0] = orb->create_policy(
LWFT::CLIENT_FAILOVER_SEMANTICS_POLICY_TYPE, failover);
CORBA::Object_var obj =
  orb->resolve_initial_references("ORBPolicyManager");
CORBA::PolicyManager_var policy_manager =
  CORBA::PolicyManager::_narrow(obj.in());
policy_manager->set_policy_overrides(policy_list, CORBA::ADD_OVERRIDE);
policy_list[0]->destroy ();


// From now on, any invocation on a group reference IOGR should contain
// FTRequestServiceContext
```

THALES

Enforcing At Most Once Semantics

Applications are responsible for enforcing at-most once semantics when required

- Use of Current object retrieved via ORB resolve_initial_references ("FTCurrent") operation
- Extract info from FT_REQUEST Service Context:
  - client_id;
  - retention_id;
  - expiration_time;

```
local interface Current : CORBA::Current {

    string get_client_id() raises(NoContext);

    long get_retention_id()

        raises(NoContext);

    TimeBase::TimeT get_expiration_time()

        raises(NoContext);

};
```

```
CORBA::Object_var obj = orb->resolve_intial_references("FTCurrent");
LWFT::Current_var ft_current = LWFT::Current::_narrow(obj.in());
...
void increment_value() {
   CORBA::String_var client_id = ft_current->get_client_id();
   CORBA::Long retention_id = ft_current->get_retention_id();

   if (already_done(client_id,retention_id))
      return;
   else
      do_increment();
}
```

THALES

Naming Object Groups

Human readable naming schemes

→ *Ease bootstrapping*

→ *Ease service localization*

```
module LWFT {
   typedef PortableGroup::Name ObjectGroupName;
   typedef sequence <ObjectGroupName>
   ObjectGroupNameSeq;

 };
```

"<server_id>.SRV/<orb_id>.ORB/<poa_id>.POA[/<sub_poa_id>.POA*]/<object_id>.OBJ "

where poa_id is the first named POA below RootPOA and sub_poa values are in order of occurrence beneath that POA.

"<server_id>/<object_id>"

Allowed shorthand notation when no ambiguity

THALES

Naming Object Groups : Support for Aliases

Aliases for application-oriented naming schemes

→ *Only with ObjectGroupManager*

```
module LWFT {

  const string FT_ALIAS = "org.omg.ft.alias";

};
```

- ***The object group name (alias) is unique within the context of the Fault Tolerance domain.***
- ***FT Property 'org.omg.ft.alias' at object creation via PortableGroup::GenericFactory interface.***
  - ***Default Alias "<object_group_id>.GID "***
- ***Administrative tools may also be used for naming the object groups via implementation-specific interfaces.***

THALES

Service Location

Obtain an ObjectGroup from an ObjectGroupName

→ ORB::resolve_initial_references (*"FTServiceLocator*").

```
interface ServiceLocator

{
PortableGroup::ObjectGroup
locate (in ObjectGroupName group)
 raises (PortableGroup::ObjectGroupNotFound);

attribute ServiceLocator fallback;

};
```

Extract the ObjectGroupName from a CORBA Request in a portable way.

- *At any PI request interception.*

```
local interface RequestDecoder

{
ObjectGroupName
get_name_from_request
(in PortableInterceptor::RequestInfo request,
out ObjectGroupName group_name);

};
```

THALES

Object Group Manager

From Portable Group (MIOP, FT CORBA, Data Parallel …)

Required for object group membership operation for IOGRs (Extended Object Group Management)

For use by the Portable Object Adapter subject to ObjectGroupMembershipPolicy.

→ ORB resolve_initial_references (“FTObjectGroupManager”)

```
module PortableGroup {

interface ObjectGroupManager {
   ObjectGroup create_member (…)
      raises  (…);

   ObjectGroup add_member (…)
      raises (…);

   ObjectGroup remove_member (…)
      raises (…);

   Locations locations_of_members (…)
     raises(…);

   ObjectGroupId get_object_group_id (…)
     raises(…);

   ObjectGroup get_object_group_ref (…)
     raises(…);

   Object get_member_ref (…)
     raises(…);
}; // end ObjectGroupManager

}; //end of PortableGroup
```

THALES

Group Object Adapter

Introduced by the MIOP specification to encapsulate object group membership.

- The GOA is created when using AUTO_OBJ_GRP_MNGR or USER_CTRL_OBJ_GRP_MNGR for ObjectGroupMembershipPolicy.
- The GOA performs automatic calls to add_member() and remove_member() on the ObjectGroupManager.

- For AUTO_OBJ_GRP_MNGR, the POA first resolves the object group name using
  - The Locator interface, or
  - The NameService when no Locator is available.

```
module PortableServer {

exception NotAGroupObject {};

typedef sequence <PortableServer::ObjectId>  IDs;

interface GOA :  ::PortableServer::POA {

    PortableServer::ObjectId
        create_id_for_reference(
            in Object the_ref)
        raises (NotAGroupObject);
    Ids reference_to_ids (
        in Object the_ref)
        raises (NotAGroupObject);
    void associate_reference_with_id (
        in Object ref,
        in PortableServer::ObjectId oid)
        raises(NotAGroupObject);
    void disassociate_reference_with_id (
        in Object ref,
        in PortableServer::ObjectId oid)
        raises(NotAGroupObject);

};

};
```

THALES

ObjectGroupMembershipPolicy POA Policy

Controls object group management in a portable way.

- **AUTO_ORB_CTRL**
  Objects created on the POA will be automatically FT replicated across all similar server processes (i.e. having a matching FT POA hierarchy).
- **AUTO_OBJ_GRP_MNGR**
  AUTO_ORB_CTRL and the POA will use the relevant ObjectGroupManager operations.
- At object reference creation the POA will make use of appropriate GOA / ObjectGroupManager operations to:
  1. Get the group to which this object should belong to
  2. Add the member to it.
  3. Return the new IOGR to the caller of the POA operation.
- **USER_CTRL_OBJ_GRP_MNGR**
  Creates a GOA. GOA object group management operations via the ObjectGroupManager.

```
module LWFT {
enum ObjectGroupMembershipPolicyValue
{
    AUTO_ORB_CTRL,
    AUTO_OBJ_GRP_MNGR,
    USER_CTRL_OBJ_GRP_MNGR
};

local interface ObjectGroupMembershipPolicy : CORBA::Policy
{
   readonly attribute ObjectGroupMembershipPolicyValue value;
};

};
```

Implementation Repository

THALES

ObjectGroupNameResolutionPolicy

Advises POA to select a fully qualified naming scheme or a shorthand notation when ObjectGroupMembershipPolicy AUTO_OBJ_GRP_MNGR value is used.

```
module LWFT {
   enum ObjectGroupNameResolutionPolicyValue
   {
         USE_FULLY_QUALIFIED_NAME,
         USE_SHORTHAND_NOTATION
   };
   local interface
   ObjectGroupNameResolutionPolicy :
   CORBA::Policy
   {
         readonly attribute
         ObjectGroupNameResolutionPolicyValue value;
   };

};
```

THALES

THALES

Hello Application

Sample Code

Hello Client

Hello Server

Warm passive replication

Hello Server

```
module Test {
   interface Hello {
      void say_hello(in string s);
   };
};
```

Simple CORBA client code!

```
int main( int argc, char* argv[] )
{

CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);

CORBA::Object_var obj = .. ; // Get Hello object reference

Test::Hello_var hello = Test::Hello::_narrow(obj.in());

Hello->say_hello("Hello World!");

orb->destroy();

}
```

Basic Hello Server

```
int main( int argc, char* argv[] )
{
CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);
CORBA::Object_var obj = orb->resolve_intial_references("FTServerManager");
LWFT::ServerManager_var sm = LWFT::ServerManager::_narrow(obj.in());

HelloImpl impl;

// And activate the root POA
obj = orb->resolve_initial_references( "RootPOA" );
PortableServer::POA_var rootPOA = PortableServer::POA::_narrow( obj.in() );

// Create PERSISTENT POA and USER_ID
PortableServer::POA_var poa = rootPOA->create_POA( … );

PortableServer::ObjectId_var oid = PortableServer::string_to_ObjectId("MyHello");
poa->activate_object_with_id(oid, &impl);

PortableServer::POAManager_var poaMgr = rootPOA->the_POAManager();
poaMgr->activate();

srm->register_server (); // Join the process group now
// Perform Primary Initialisation or Backup Insertion
LWFT::ServerReplicationRoleValue role = srm->replica_role();
if (role == LWFT::SRV_REPL_PRIMARY) {
   // Perform Primary Initialisation
} else if (role == LWFT::SRV_REPL_BACKUP) {
   // Perform Backup Insertion
}
srm->server_ready (); // Server replica is ready now.
orb->run(); // Enter in the main loop
```

```
class HelloImp : public virtual
POA_Test::Hello
{
public:

 void say_hello(const char* s)
 {
 }

};
```

```
Host1> ./server –ORBServerId HelloAPP –ORBFTLocation HOST1
```

```
Host2> ./server –ORBServerId HelloAPP –ORBFTLocation HOST2
```


THALES

Advanced Hello Client

```
int main( int argc, char* argv[] )
{

CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);

// Create the ClientFailoverSemanticsPolicy
CORBA::Any failover;
failover <<= LWFT::AT MOST ONCE;

CORBA::PolicyList policy_list;
// Set the policy
policy_list.length(1);
policy_list[0] = orb->create_policy(
LWFT::CLIENT_FAILOVER_SEMANTICS_POLICY_TYPE, failover);
CORBA::Object_var obj =
  orb->resolve_initial_references("ORBPolicyManager");
CORBA::PolicyManager_var policy_manager =
  CORBA::PolicyManager::_narrow(obj.in());
policy_manager->set_policy_overrides(policy_list, CORBA::ADD_OVERRIDE);
policy_list[0]->destroy ();


// From now on, any invocation on a group reference IOGR should contain
// FTRequestServiceContext


CORBA::Object_var obj = .. ; // Get Hello object reference

Test::Hello_var hello = Test::Hello::_narrow(obj.in());

Hello->say_hello("Hello World!");

orb->destroy();

}
```

Advanced Hello Server: LwFT aware ORB

```
class HelloImp : public virtual POA_Test::Hello
{
 LWFT::FTCurrent_var current_;
public:
 HelloImp(CORBA::ORB_ptr orb);
 void say_hello(const char* s);
};

HelloImpl::HelloImp(CORBA::ORB_ptr orb)
{
   CORBA::Object_var obj =  orb->resolve_initial_referencs("FTCurrent);
   current_ = LWFT::FTCurrent::_narrow(obj.in());
}

void HelloImpl::say_hello(const char* s)
{
  CORBA::String_var client_id =  current_->get_client_id();
  CORBA::Long retention_id =  current_->get_retention_id();
  if (already_done(client_id,retention_id))
      return;
  else
      do_say(s)();
}
```

Advanced Hello Server: LwFT aware ORB (1/2)

```
int main( int argc, char* argv[] )
{
CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);

HelloImpl impl(orb.in());

// And activate the root POA
CORBA::Object_var obj = orb->resolve_initial_references( "RootPOA" );
PortableServer::POA_var rootPOA = PortableServer::POA::_narrow( obj.in() );

PortableServer::POAManager_var poaMgr = rootPOA->the_POAManager();

// Create policy list for object_group_membership_policy and persistence
CORBA::PolicyList pl(3);
pl.length(3);
pl[0]=rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
pl[1]=rootPOA->create_id_assignment_policy(PortableServer::USER_ID);
Pl[2]=rootPOA->create_object_group_membership_policy(LWFT::USER_CTRL_OBJ_GRP_MNGR);

PortableServer::POA_var poa = rootPOA->create_POA( "FTPOA", poaMgr.in(), pl);
PortableGroup::GOA_var goa = PortableGroup::GOA::_narrow(poa.in());

CORBA::Object_var groupRef = … ; // Retrieve Group reference
poaMgr->activate();

obj = orb->resolve_intial_references("FTServerManager");
LWFT::ServerManager_var sm = LWFT::ServerManager::_narrow(obj.in());

sm->register_server (); // Join the process group now

// join object group
PortableServer::ObjectId_var oid = goa->create_id_for_reference(groupRef.in());
goa->associate_reference_with_id (groupRef.in(), oid.in());
goa->activate_object_with_id(oid, &impl);
```

ObjectGroupMembershipPolicy

Retrieval of ServerManager

Joining Server Group

-ORBServerID

Object Group membership operation à la MIOP

THALES

Advanced Hello Server: LwFT aware ORB (2/2)

```
// Perform Primary Initialisation or Backup Insertion
LWFT::ServerReplicationRoleValue role = srm->replica_role();
if (role == LWFT::SRV_REPL_PRIMARY) {
   // Perform Primary Initialisation
} else if (role == LWFT::SRV_REPL_BACKUP) {
   // Perform Backup Insertion
}
sm->server_ready (); // Server replica is ready now.

orb->run(); // Enter in the main loop

goa->disassociate_reference_with_id (memberRef.in(),oid.in());
sm->unregister_server();

if (!CORBA::is_nil(orb)) {
  try {
    orb->destroy();
  } catch (const CORBA::Exception &) {
   …
}

}
```

Application-specific state synchronization

Server is ready for service

Object Group membership operation à la MIOP

Graceful stop

```
Host1> ./server –ORBServerId HelloAPP –ORBFTLocation HOST1
```

```
Host2> ./server –ORBServerId HelloAPP –ORBFTLocation HOST2
```

Advanced Hello Server: NON LwFT aware (1/2)

```
int main( int argc, char* argv[] )
{
CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);

HelloImpl impl(orb.in());

// And activate the root POA
CORBA::Object_var obj = orb->resolve_initial_references( "RootPOA" );
PortableServer::POA_var rootPOA = PortableServer::POA::_narrow( obj.in() );

PortableServer::POAManager_var poaMgr = rootPOA->the_POAManager();

// Create policy list for object_group_membership_policy and persistence
CORBA::PolicyList pl(2);
pl.length(2);
pl[0]=rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
pl[1]=rootPOA->create_id_assignment_policy(PortableServer::USER_ID);

PortableServer::POA_var poa = rootPOA->create_POA( "FTPOA", poaMgr.in(), pl);

obj = orb->resolve_initial_references( "FTObjectGroupManager" );
PortableGroup::ObjectGroupManager_var ogm = PortableGroup::ObjectGroupManager::_narrow(obj.in());

CORBA::Object_var groupRef = … ; // Retrieve Group reference
Std::string rep_id = " …" ; // repository ID

poaMgr->activate();

obj = orb->resolve_intial_references("FTServerManager");
LWFT::ServerManager_var sm = LWFT::ServerManager::_narrow(obj.in());

sm->register_server (); // Join the process group now
```

Ordinary server

Retrieval of ObjectGroupManager

Retrieval of ServerManager

Joining Server Group

-ORBServerID

THALES

Advanced Hello Server: NON LwFT aware (2/2)

```
PortableServer::ObjectId_var oid = PortableServer::string_to_ObjectId("MyHello");
CORBA::Object_var member = poa->create_reference_with_id(oid.in(), rep_id.c_str());

groupRef = ogm->add_member(groupRef.in(),member.in(), sm->the_location () ); // jo

poa->activate_object_with_id(oid, &impl);

// Perform Primary Initialisation or Backup Insertion
LWFT::ServerReplicationRoleValue role = srm->replica_role();
if (role == LWFT::SRV_REPL_PRIMARY) {
   // Perform Primary Initialisation
} else if (role == LWFT::SRV_REPL_BACKUP) {
   // Perform Backup Insertion
}
sm->server_ready (); // Server replica is ready now.

orb->run(); // Enter in the main loop

ogm->remove_member(groupRef.in(), sm->the_location () );
sm->unregister_server();

if (!CORBA::is_nil(orb)) {
  try {
    orb->destroy();
  } catch (const CORBA::Exception &) {
   …
}

}
```

Explicit Object Group membership à la FT CORBA

Application-specific state synchronization

Server is ready for service

Explicit Object Group membership à la FT CORBA

Graceful stop

```
Host1> ./server -ORBServerId HelloAPP -ORBFTLocation HOST1
```

```
Host2> ./server -ORBServerId HelloAPP -ORBFTLocation HOST2
```


THALES

Conclusion

- Server Replica management is not CORBA-dependent
- Basic FT applications are just ordinary CORBA applications
- IOGRs borrowed from FT CORBA
  - Failure and replication transparencies
  - Help in enforcing at most once semantics
- Full interoperability
  - Client/Server interactions across heterogeneous ORB
  - Replicas on different ORBs
- Reuse of standards
  - Subset of FT CORBA
    - ReplicationManager, IOGR, Transport heartbeats
    - Well defined failover semantics
  - CORBA Messaging policies
  - RT CORBA
  - Group Object Adapter (GOA) and PortableGroup

For more information, contact:

Hakim SOUAMI
Thales Air Systems
Phone: +33 (0) 1 79 61 21 80
E-mail: hakim.souami@thalesgroup.com

Reference implementation soon to be released

CARDAMOM, LGPL, Q3 2012

Questions?