Fault Tolerant CORBA Tutorial

OMG Workshop on

Embedded & Real-time

Distributed Object Systems

Priya Narasimhan
Carnegie-Mellon Univ.
Eternal Systems, Inc.

Joey Garon
Vertel Corporation

Fault Tolerant CORBA References

Download specification (chapter 25 of CORBA 2.5) from

- **ftp://ftp.omg.org/pub/docs/formal/01-09-29.pdf**

Download this tutorial from

- **http://www.eternal-systems.com**
- **http://www.vertel.com**
- **http://www.cs.cmu.edu/~priya**

Contact us at

- **priya@cs.cmu.edu**
- **joey-garon@vertel.com**

Outline

1. **Introduction to Fault Tolerance**
2. Fault Tolerance mechanisms
3. Fault Tolerance properties

   Break

4. Fault Tolerance management
5. Fault Tolerance applications
6. Fault tolerant hello server example

The Need for Fault Tolerance

- Hardware sometimes fails
- Many systems must run continuously
  - Health, safety, financial reasons
    - Air traffic control and defense systems
    - 911 and medical systems
    - Aircraft instrumentation, manufacturing control systems, telephony and networking systems
- Cannot completely eliminate possibility of failure
- Reduce possibility—No single point of failure

Cost of Application Downtime

Industry	Business Operation Industry	Average Cost per Hour of Downtime
Financial	Brokerage operations	$6.5 million
Financial	Credit card/sales authorization	$2.6 million
Media	Pay-per-view television	$1.1 million
Retail	Home shopping (TV)	$113.0 thousand
Retail	Home catalog sales	$90.0 thousand
Transportation	Airline reservations	$89.5 thousand

Gartner Group report 1998

Standish Group Research Note Copyright 1999

Application	Cost of Downtime / Minute @ Normal Load	Cost of Downtime / Minute @ Peak Load
Customer Relationship Management	**$2,200**	**$2,500**
Data Warehouse	**$5,800**	**$6,300**
Electronic Commerce	**$2,500**	**$7,800**
ERP	**$6,400**	**$7,900**
Supply Chain Mgmt	**$4,400**	**$6,500**

Standish Group Research Note Copyright 1999

No Single Point of Failure

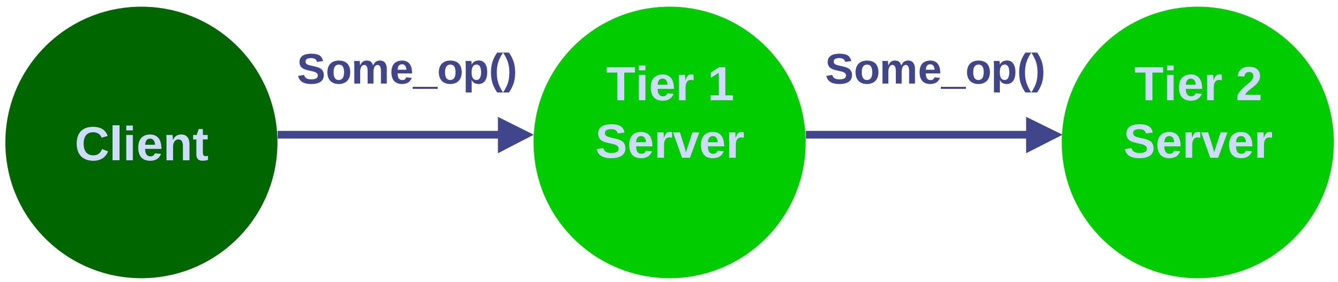
- Redundancy (replication)
  - Creating and distributing multiple copies of an object/process
- Fault detection
  - Discovering that a processor/process/object has failed
- Recovery
  - Re-instating a failed processor/process/object to normal operational status

Strong replica consistency!

Types of Faults

- **Crash faults**
  - Hardware and/or OS crashes in isolation
  - Process and/or Object crashes
- **Communication faults**
  - Message loss
  - Network partitioning
- **Malicious faults (commission/byzantine)**
  - Processor/process/object maliciously subverted
- **Omission faults**
  - Missed deadline in a real-time system
- **Design faults**
  - Correlated software/programming/design errors

CORBA without FT



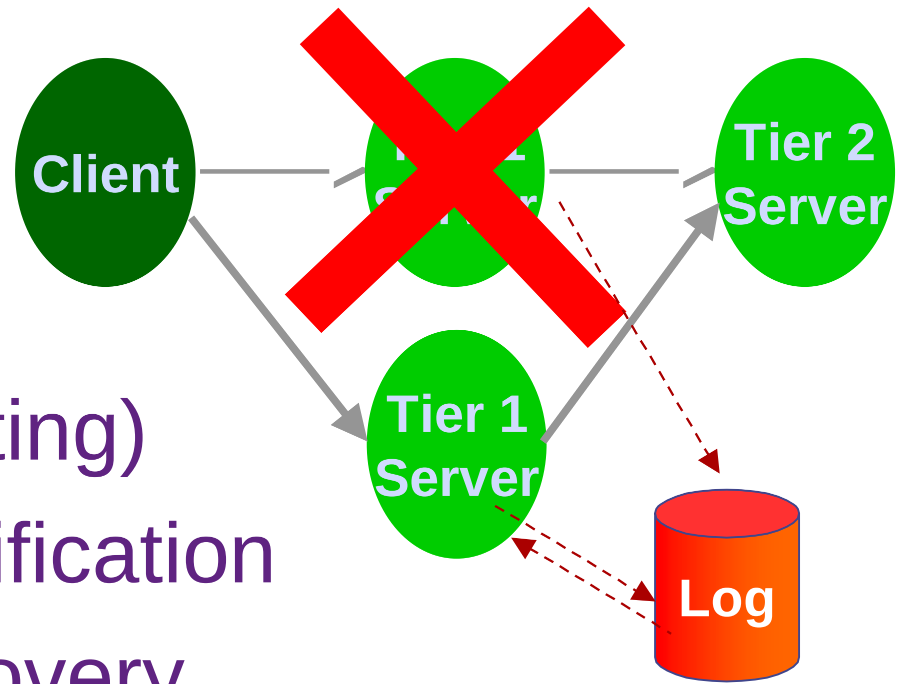
- Core CORBA does not specify mechanisms for redundancy or failover
- TCP/IP alone does not allow rapid detection of faults

CORBA FT Framework

- Minimum modification to applications
  - Transparent replication and recovery
- A Framework for Fault Tolerance, supports
  - wide range of app types
  - multiple FT strategies
  - wide range of application control
  - wide range of mechanisms for detection, notification, and analysis

How Fault Tolerance Works

- Replicated objects
- Fault detection
- Logging (checkpointing)
- Fault analysis & notification
- State transfer & recovery
- Client failover



Redundancy

- Object is the unit of redundancy in the standard
- Strong replica consistency
  - All of the replicas have the same state
  - Greatly simplifies the application system design
  - Requires careful design of, and strong mechanisms in, the infrastructure

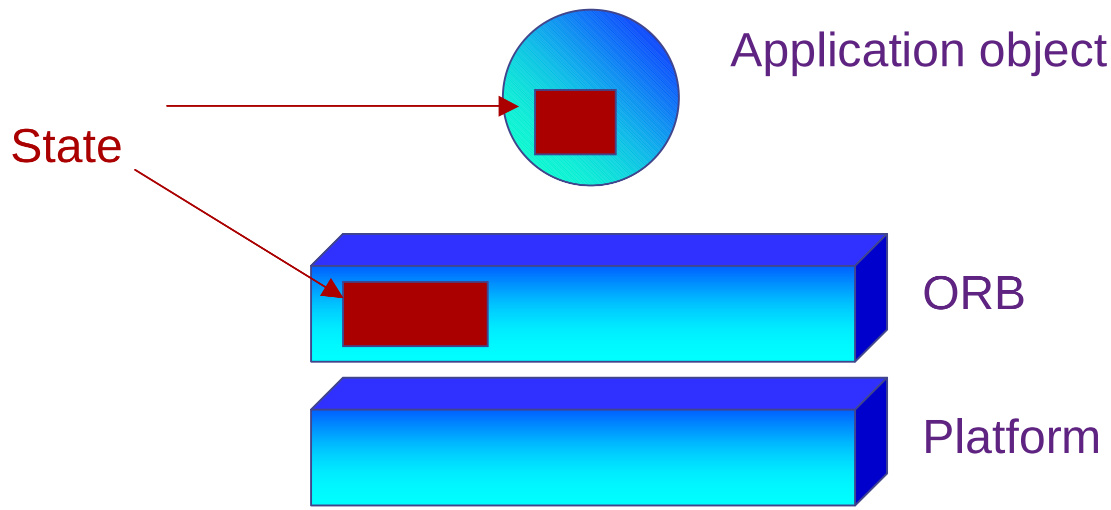
Object Groups

- **Replicas of an object form an object group**
- **Each object group has an Interoperable Object Group Reference (IOGR)**
- **Object group abstraction provides**
  - Replication transparency
  - Failure transparency

Identity Model

- CORBA supports a weak identity model
- Fault Tolerant CORBA requires
  a strong identity model
- Object groups identified by
  - FTDomainId, ObjectGroupId
- Members of object groups identified by
  - FTDomainId, ObjectGroupId, Location

What is State?



- State = values of data that must be maintained identical across all replicas of an object
- State exists both in the application and the ORB

Stateless and Stateful Systems

Stateless

- Static data and read-only operations
  - Resolve()
- Completely persist to remote store after each “write” op

Stateful

- At least one “write” op
  - Bind()
- Behavior of later ops depends on “write” op
- Typical system

Determinism

- **Determinism**
  - Two replicas, given the same sequence of input messages, arrive at the same state
- **Examples of non-determinism**
  - Local timers, I/O to local devices, “FT-unsafe” multi-threading
- **Why is determinism necessary?**
  - Assures consistent state when replicas are distributed

Replication Styles

- **You can configure Fault Tolerance to suit your application's needs**
- **Passive Replication**
  - One replica (primary) does all the work
  - Other replicas serve as backups
- **Active Replication**
  - All replicas do all the work

Fault Tolerance for the Server

- **Object Replication**
- **Object Group Properties**
  - Property Manager interface
- **Creating Fault-tolerant Objects**
  - Generic Factory interface
  - Object Group Manager interface
- **Detecting Faults**
- **State Transfers**

Fault Tolerance for the Client

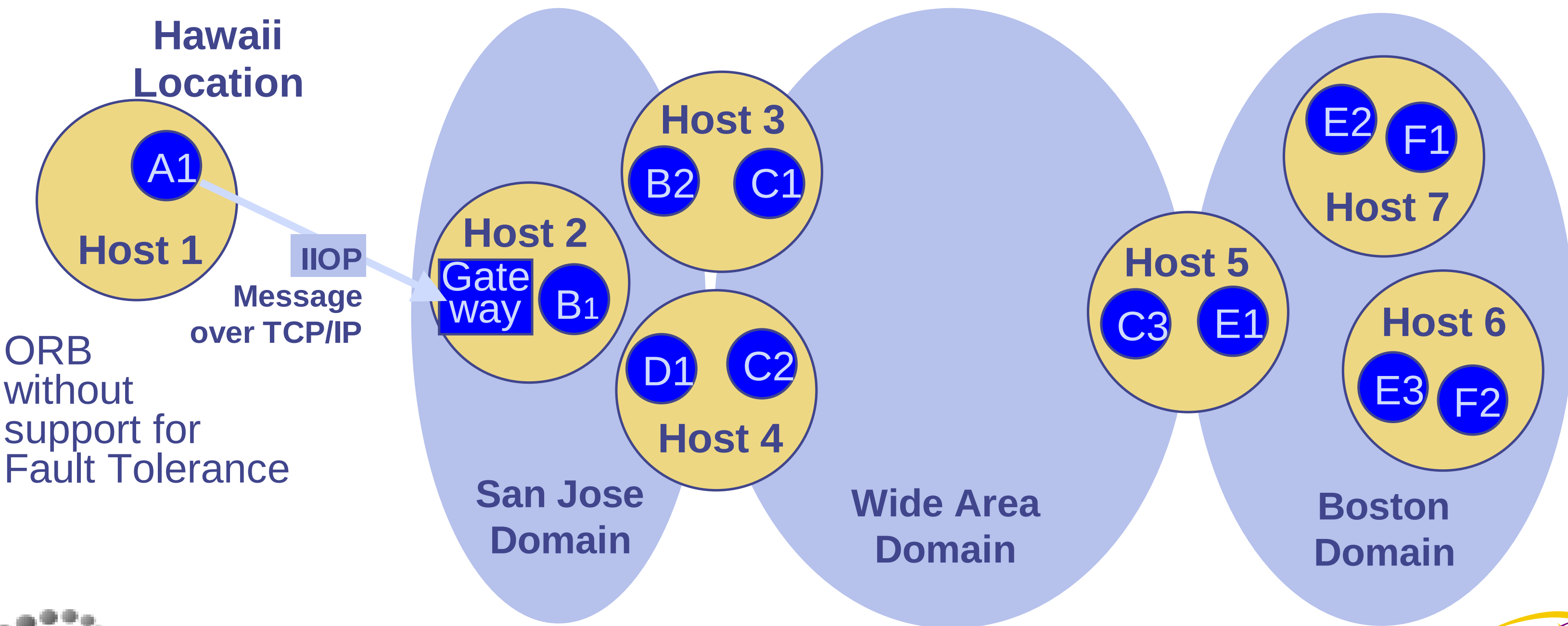
- **Failover**
  - If Server does not respond, Client should try again using the same or an alternate address
  - If Client transmits its request more than once, it should not be executed more than once
- **Addressing**
  - If Client uses an obsolete address, Server should supply an up to date address
- **Loss of Connection**
  - If Client's connection to Server fails, the Client's ORB should be informed prompty

Who Has Control?

- **Infrastructure-controlled fault tolerance**
  - Automatic creation and allocation of replicas
  - Automatic maintenance of replica consistency
  - More sensible for complex programs on servers
- **Application-controlled fault tolerance**
  - Precise control over object creation and allocation
  - Application algorithms maintain replica consistency
  - May be necessary for embedded systems

Fault Tolerance Domains

- Aid application management and provide for scalability
- Each Fault Tolerance Domain is managed by a single Replication Manager



Outline

1. Introduction to Fault Tolerance
2. Fault Tolerance mechanisms
3. Fault Tolerance properties

   Break

4. Fault Tolerance management
5. Fault Tolerance applications
6. Fault tolerant hello server example

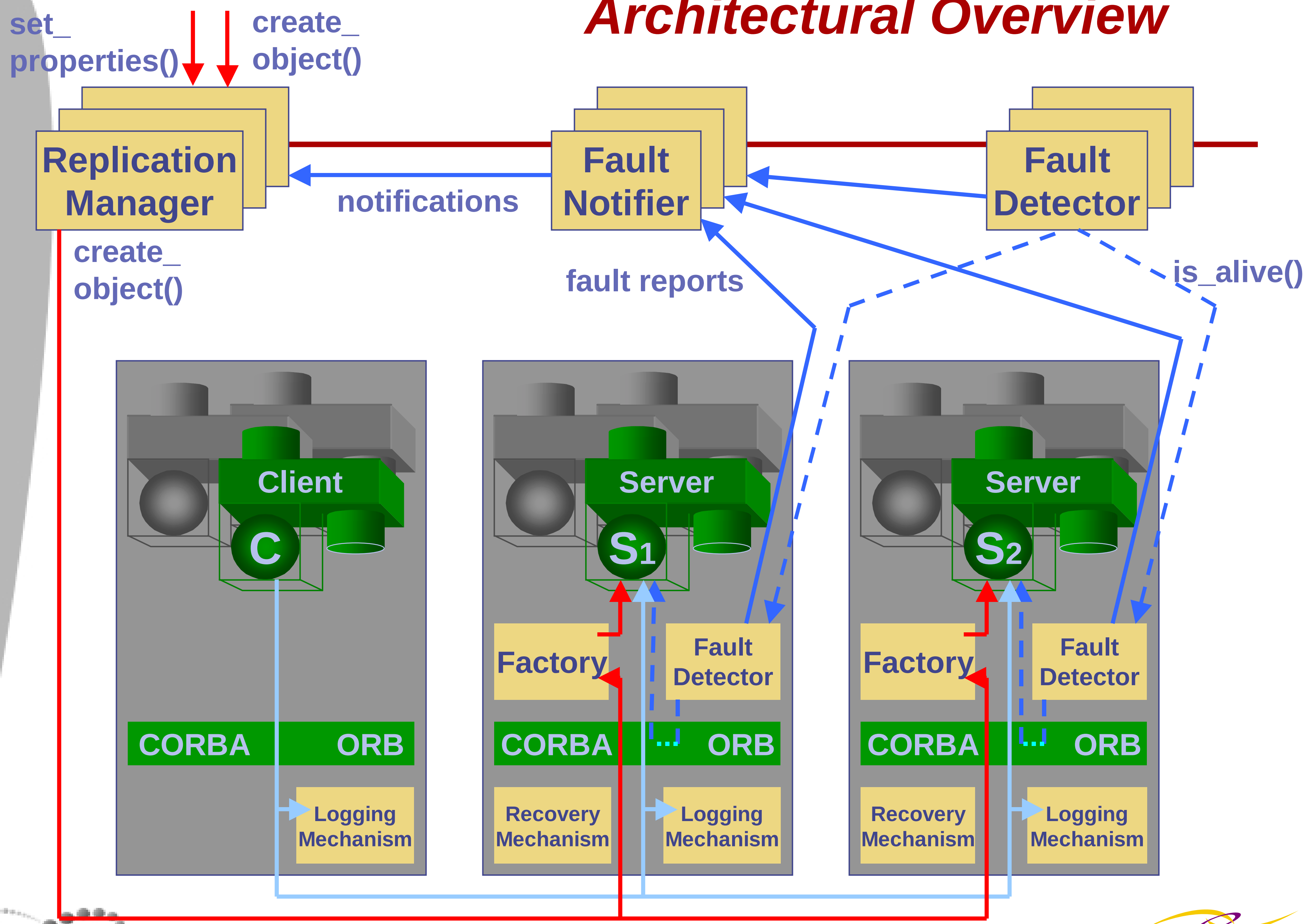
Objectives of FT CORBA

- **Wide range of fault tolerance**
  - Simple low-cost clients
  - Highly reliable server clusters
  - Many systems will contain both
  - Other systems will contain external clients that know nothing, or little, about fault tolerance
- **Local Clusters and also Wide-area Systems**
- **Large-scale Servers and also Embedded Controllers**

Limitations of the Specification

- No way to enforce determinism
- No support for
  - Partitioned systems
  - Malicious faults
  - Software design faults
- Interoperability limitations
  - All replicas of an object must be hosted by infrastructure from the same vendor
- Vendors can provide proprietary products that overcome these limitations

Architectural Overview



Outline

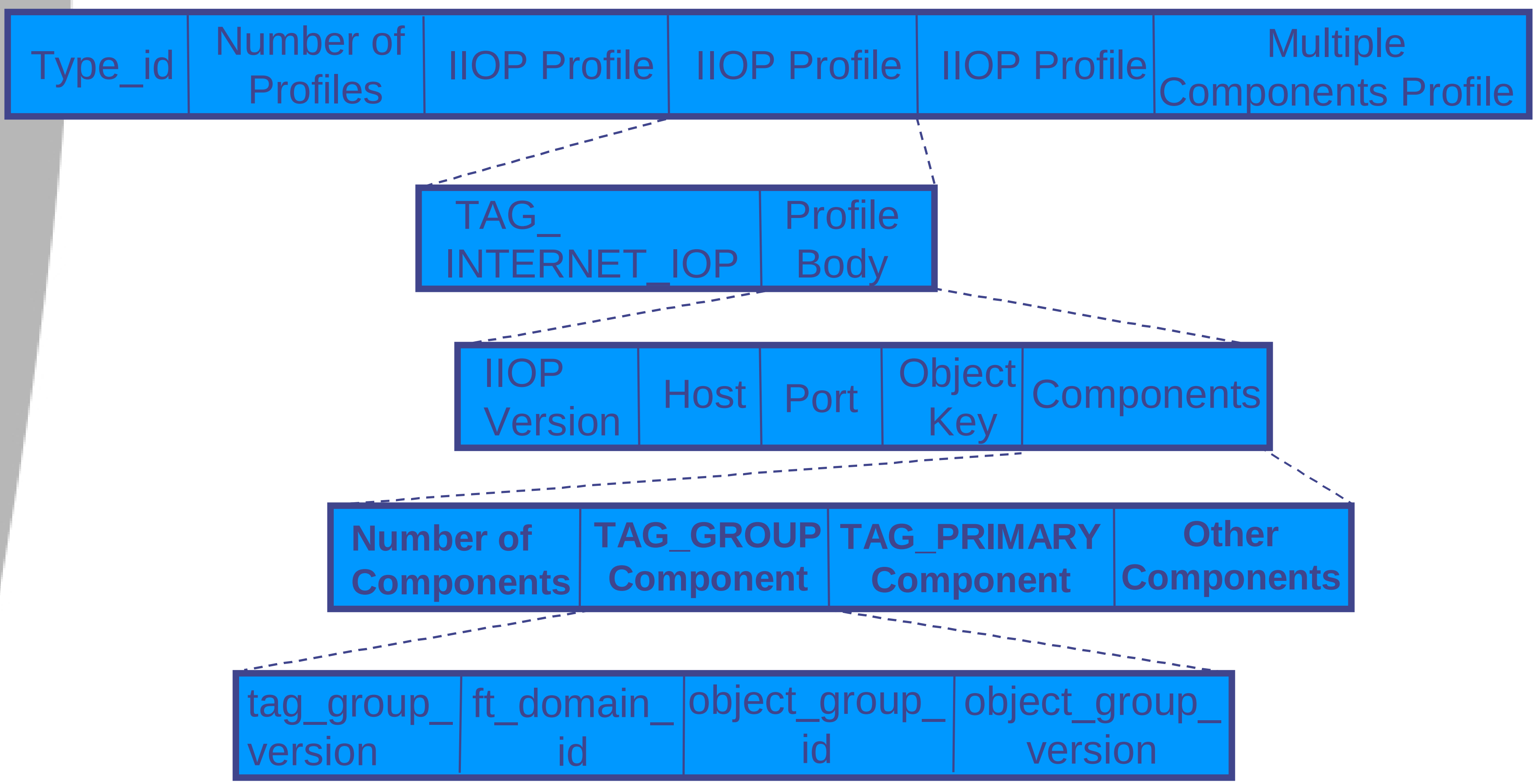
2. Fault Tolerance Mechanisms

- **a. Addressing**
- b. Failover

Interoperable Object Group Reference (IOGR)

- An IOGR is a multiple profile IOR
- Each profile contains a TAG_GROUP component, consisting of
  - FTDomainId
  - ObjectGroupId
  - ObjectGroupRefVersion
- At most one profile may contain a TAG_PRIMARY component, which gives a hint as to which profile corresponds to the primary

Interoperable Object Group Reference

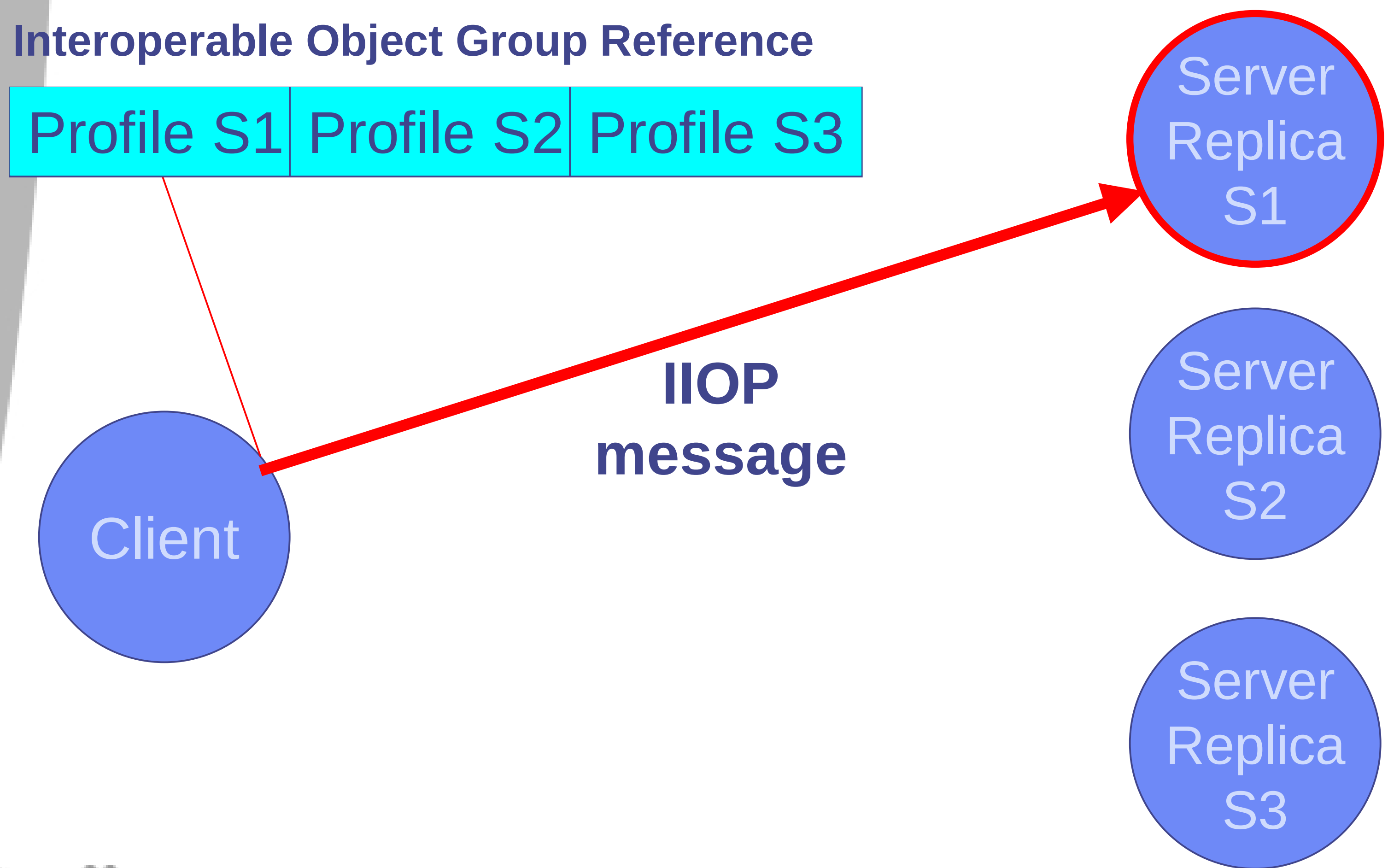


Profiles Address Object Group Members

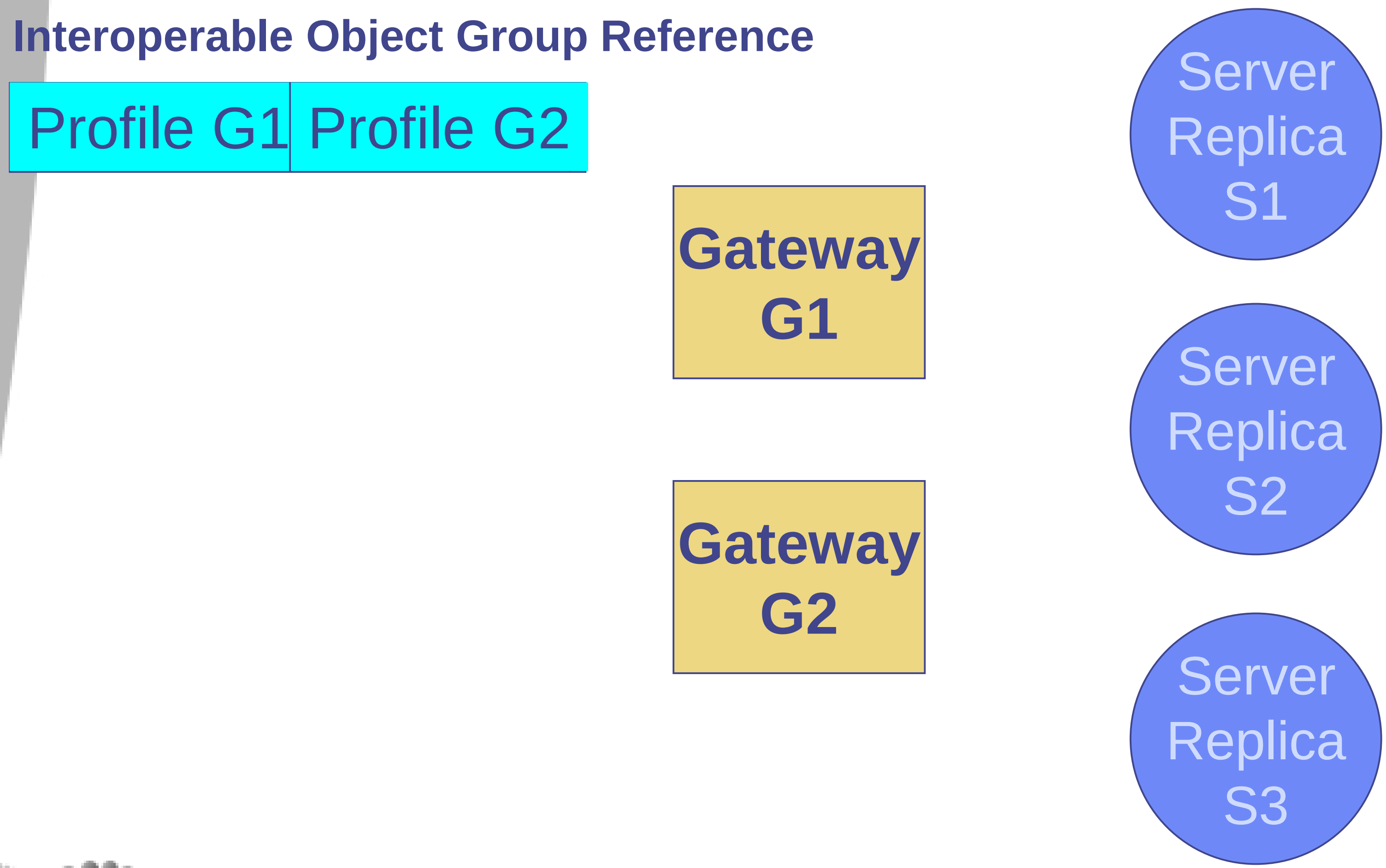
Interoperable Object Group Reference

Profile S1	Profile S2	Profile S3

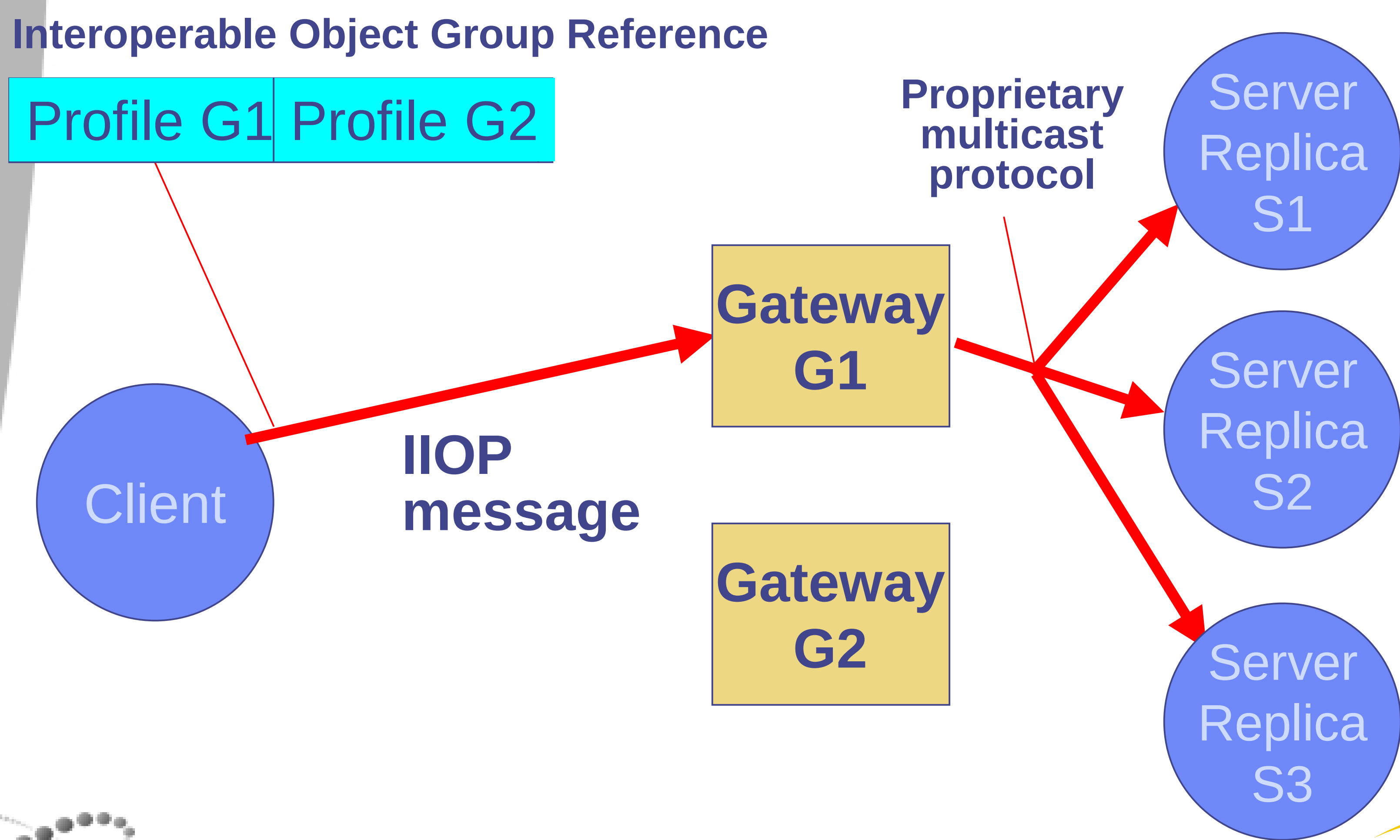
Access via IIOP Directly to Primary



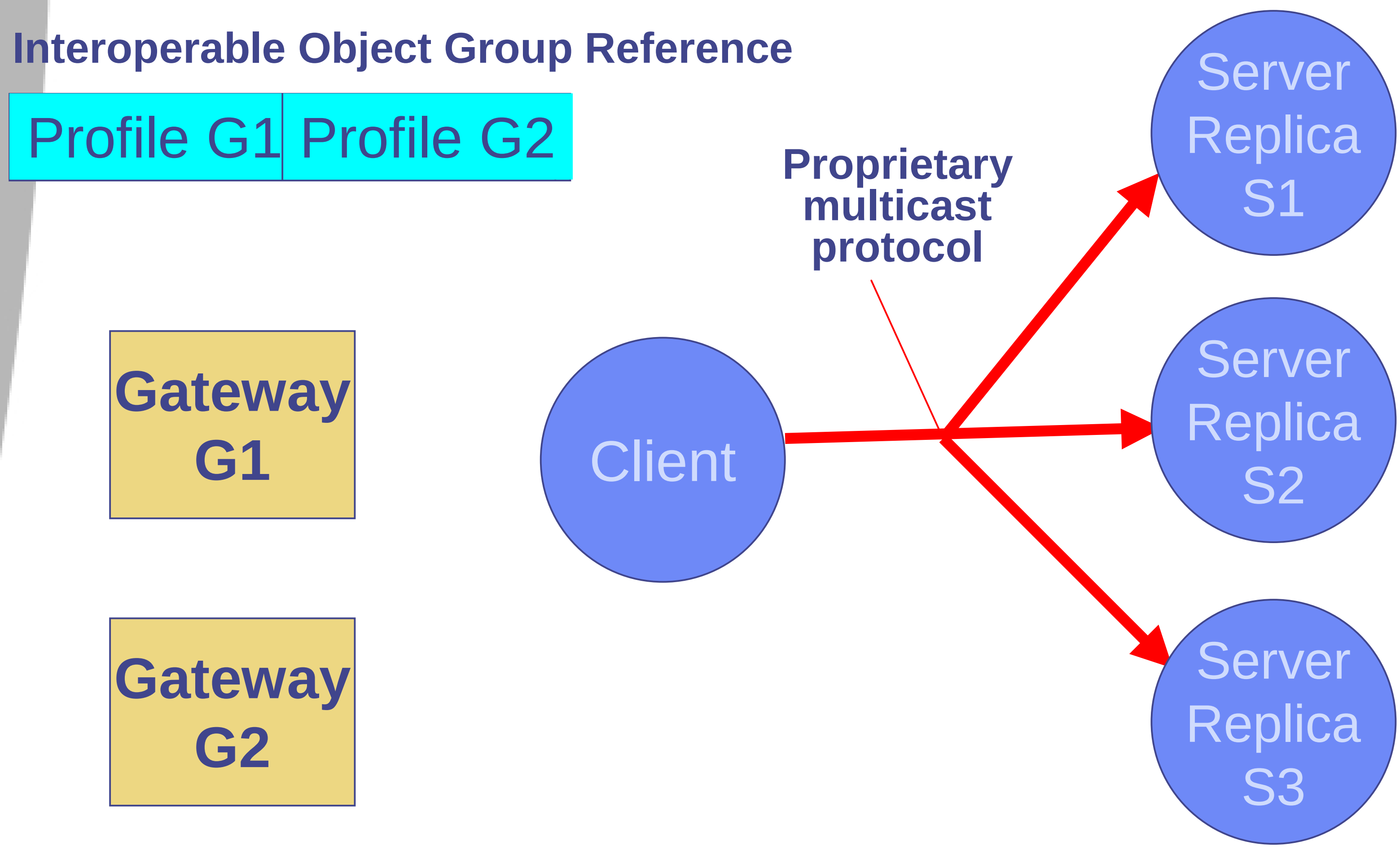
Profiles Address Gateways



Access via IIOP and a Gateway



Direct Access via a Proprietary Multicast Protocol



Most Recent Object Group Reference

- **Problem**

  Object Group Reference may not correspond to current membership of the server object group

- **Solution**

  GROUP_VERSION Service Context

  TAG_GROUP component of IOGR contains Group Version Number (GVN) for the server object group

  Client ORB puts GVN in the GROUP_VERSION Service Context of the client's request message for the server object group

Most Recent Object Group Reference

- Server ORB extracts the GVN from the request message
- If server GVN = GVN from client
  - Primary: Process request
  - Backup: Log request
- If server GVN > GVN from client
  - Throw LOCATE_FORWARD_PERM with IOGR
- If server GVN < GVN from client
  - Get new IOGR from ReplicationManager

Outline

2. Fault Tolerance Mechanisms

a. Addressing

b. Failover

Failover Semantics with Fault Tolerance

Permitted Failover Conditions

Completion Status	CORBA Exception
COMPLETED_NO **COMPLETED_MAYBE**	COMM_FAILURE TRANSIENT NO_RESPONSE OBJ_ADAPTER

Transparent Reinvocation

- **Problem**

  With reinvocation for COMPLETED_MAYBE, at-most-once semantics might be violated if no extra mechanisms are in place

- **Solution**

  REQUEST Service Context
  - Client Id
  - Retention Id
  - Expiration Time

  Allows server ORB to recognize that a request is a repetition of a previous request

  If it is, server does not re-execute the request but returns the reply that was previously generated

Transport Heartbeats

- **Problem**
  - Host or connection fails during client invocation
  - TCP/IP connection not cleanly torn down and Client ORB hangs on the connection
- **Solution**
  - Periodic heartbeat messages over the connection

Transport Heartbeats

Client Side

- HeartbeatPolicy
  - Heartbeat – On/Off
  - Heartbeat Interval
  - Heartbeat Timeout
- If profile has TAG_HEARTBEAT_ ENABLED set to true,
  - Client can set HeartbeatPolicy values
  - Client ORB invokes _FT_HB() on server

Server Side

- TAG_HEARTBEAT_ ENABLED component in profile
- HeartbeatEnabledPolicy allows server to turn heartbeats on and off
- Server ORB responds to _FT_HB()

Outline

1. Introduction to Fault Tolerance
2. Fault Tolerance mechanisms
3. Fault Tolerance properties

   Break

4. Fault Tolerance management
5. Fault Tolerance applications
6. Fault tolerant hello server example

Outline

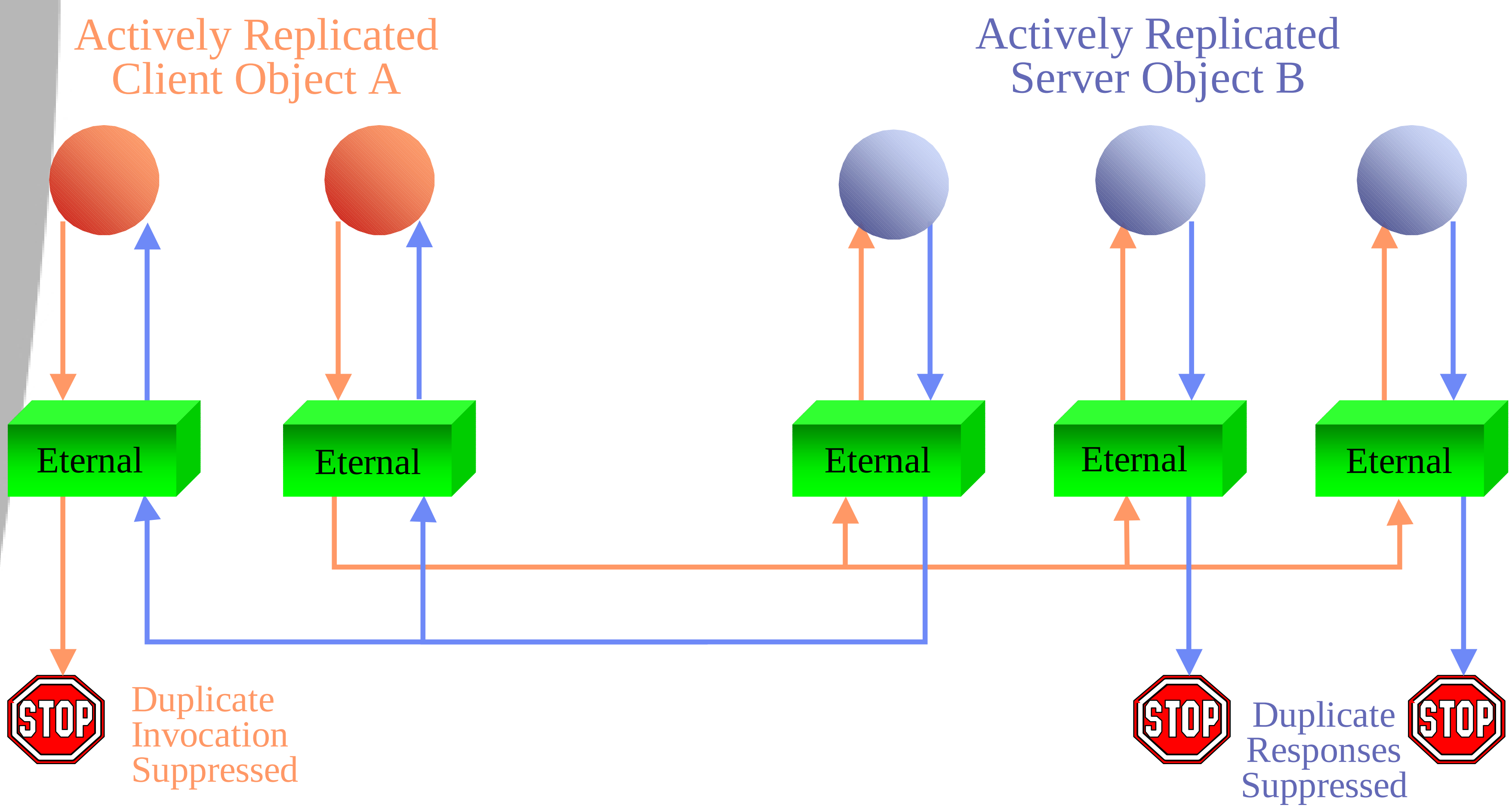
3. Fault Tolerance Properties

a. Replication Style
b. Membership Style
c. Consistency Style
d. Fault Monitoring Style
e. Fault Monitoring Granularity
f. Factories
g. Initial Number of Replicas
h. Minimum Number of Replicas
I. Fault Monitoring Interval and Timeout
j. Checkpoint Interval

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

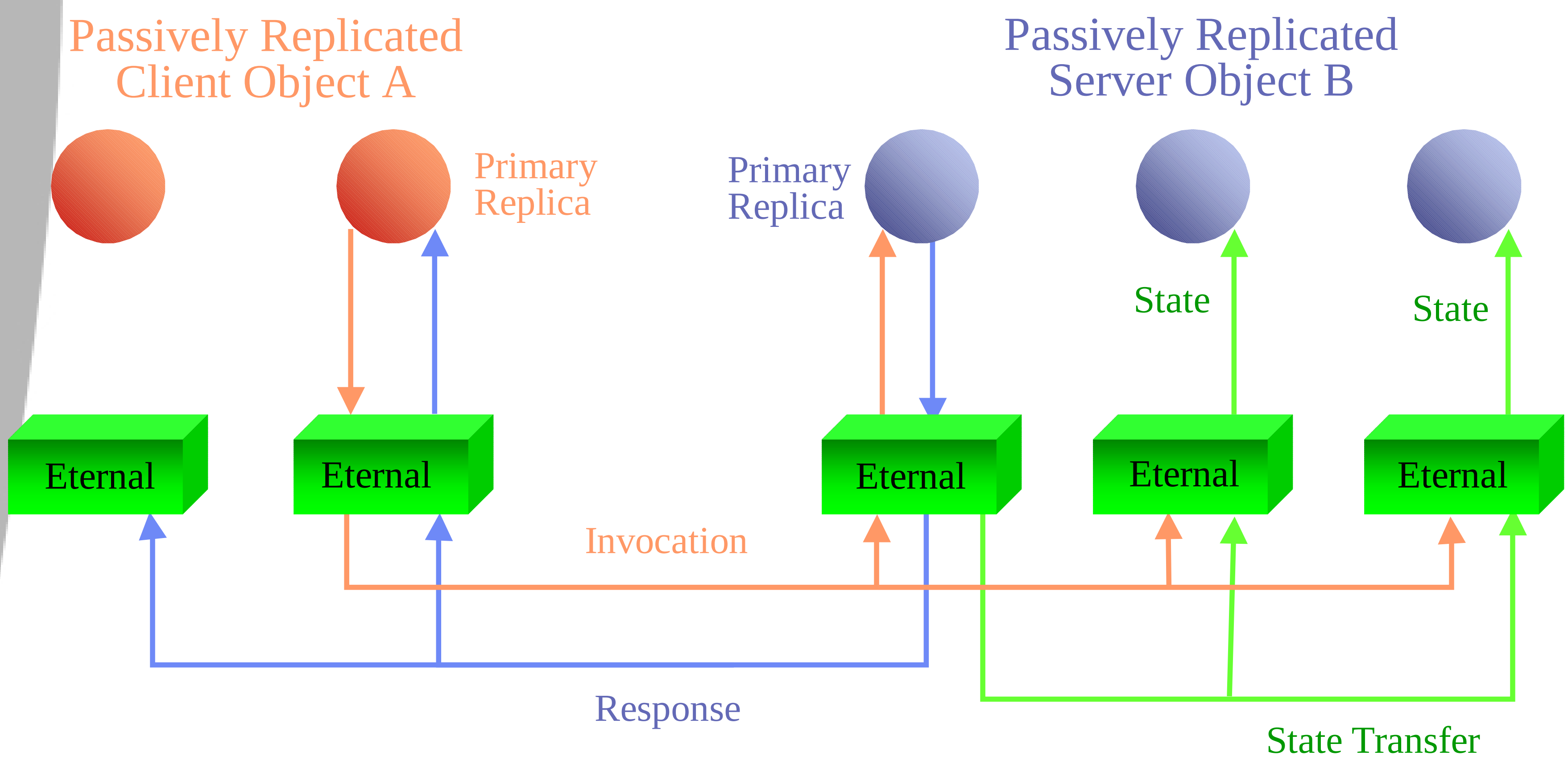
Active Replication



Active Replication

- Fault-free Conditions
  - Potential for duplicate invocations and responses
- Recovery Conditions
  - Failure of a replica is transparent to other replicas
  - State transfer to new or recovering replica
  - Queueing of messages at recovering replica for delivery after state transfer

Passive Replication



Passive Replication

- Fault-free Conditions
  - State transfer to backup replicas (warm passive only)
  - Logging of state and messages
- Recovery Conditions
  - Failure of primary requires election of, and transfer of state to, a new primary replica
  - State transfer to new or recovering warm backup replicas
  - Queueing of messages at new primary replica for delivery after state transfer
  - Potential for duplicate messages

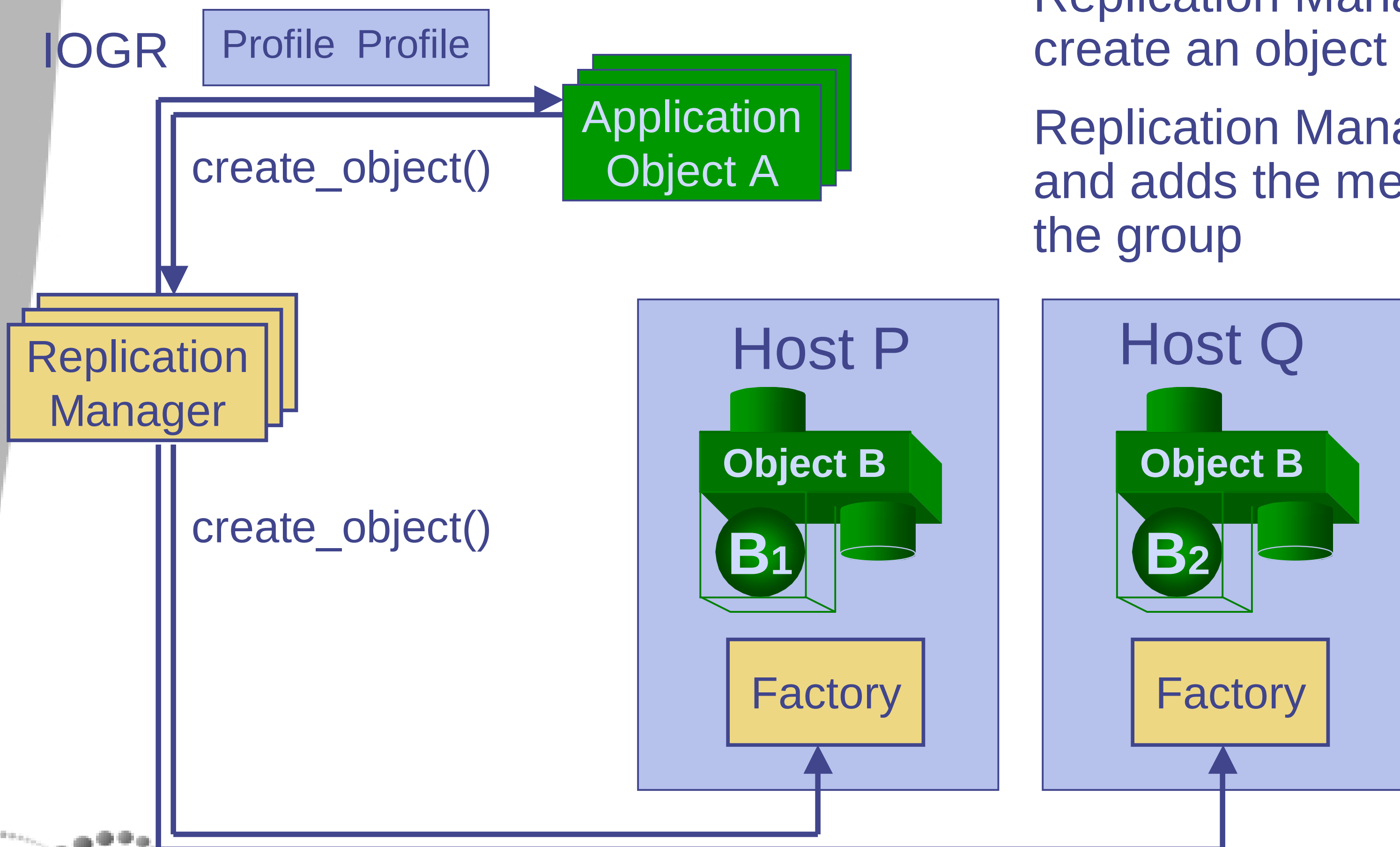
Comparison of Replication Styles

- **Passive Replication**
  - Lower memory and processing costs
  - Slower recovery from faults
- **Active Replication**
  - More memory and processing costs
  - Fastest recovery from faults
- **Underlying mechanisms are the same for both**

Membership Style

- Infrastructure-Controlled
  - Fault Tolerance Infrastructure creates multiple replicas of an object (members of an object group) and allocates them to appropriate hosts
- Application-Controlled
  - The application determines when and how many replicas to create and the hosts on which they should be created

Infrastructure-Controlled Membership Style

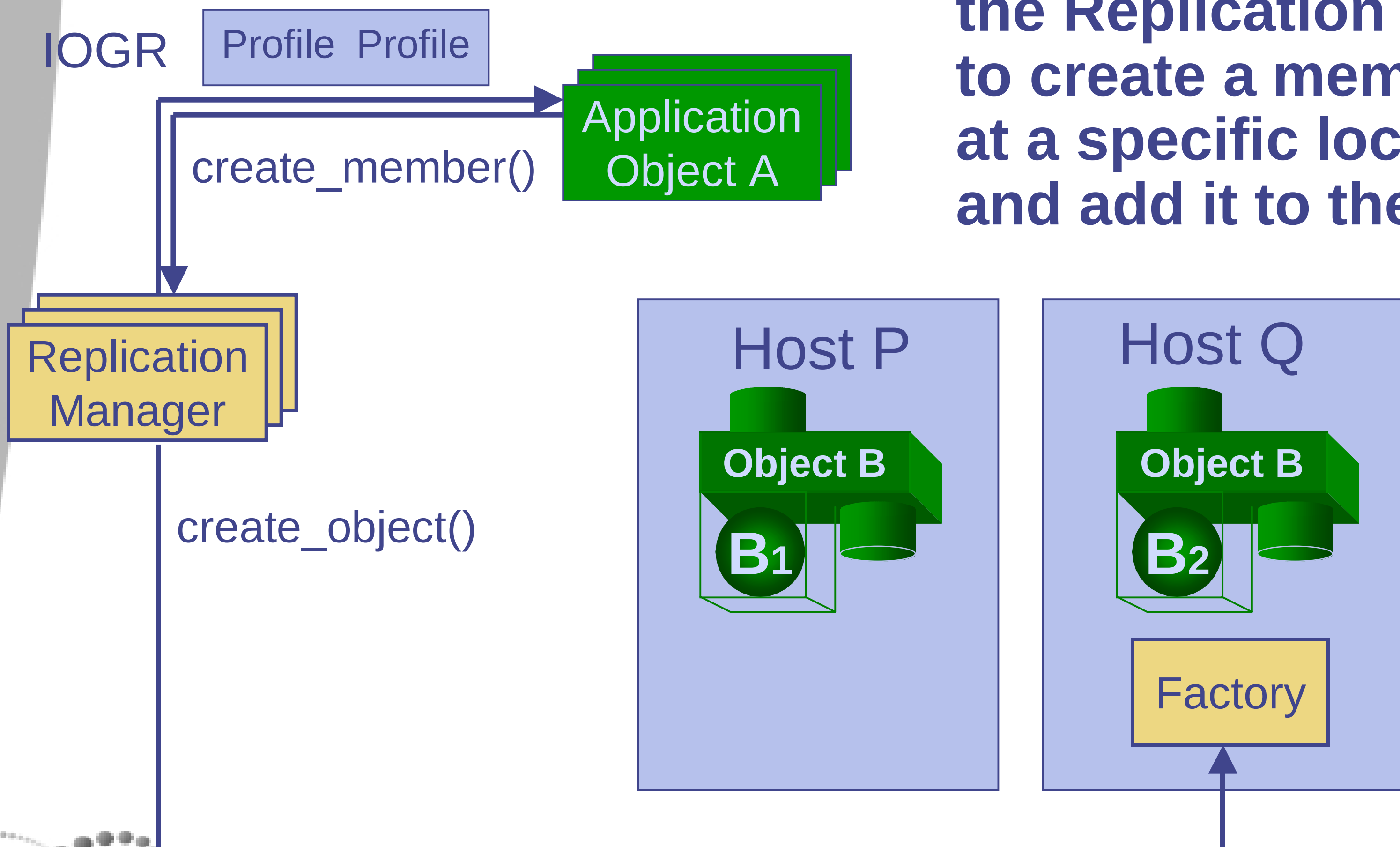


Application directs Replication Manager to create an object group

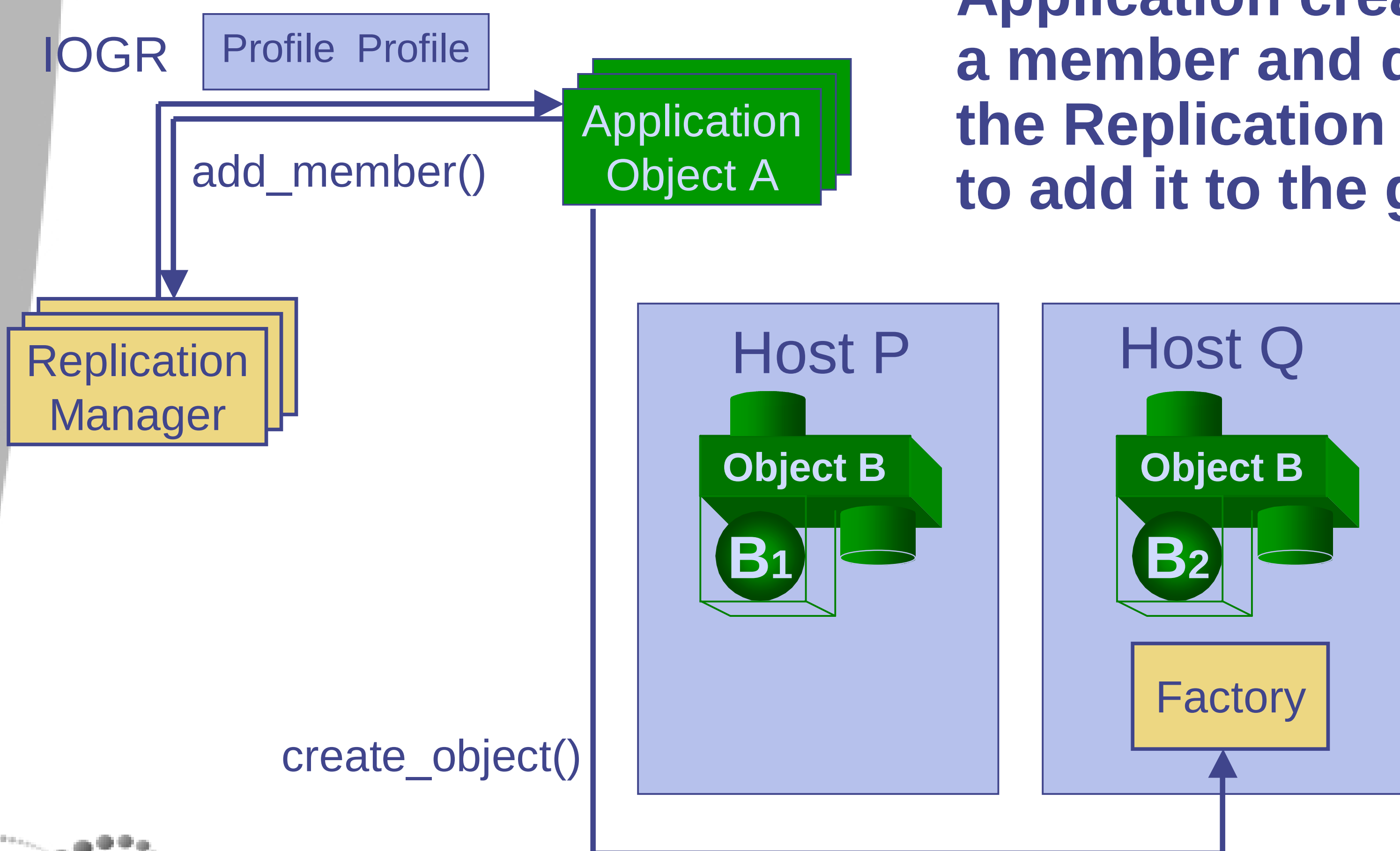
Replication Manager creates and adds the members to the group

Application-Controlled Membership Style

Application directs the Replication Manager to create a member at a specific location and add it to the group



Application-Controlled Membership Style



Application creates a member and directs the Replication Manager to add it to the group

Consistency Style

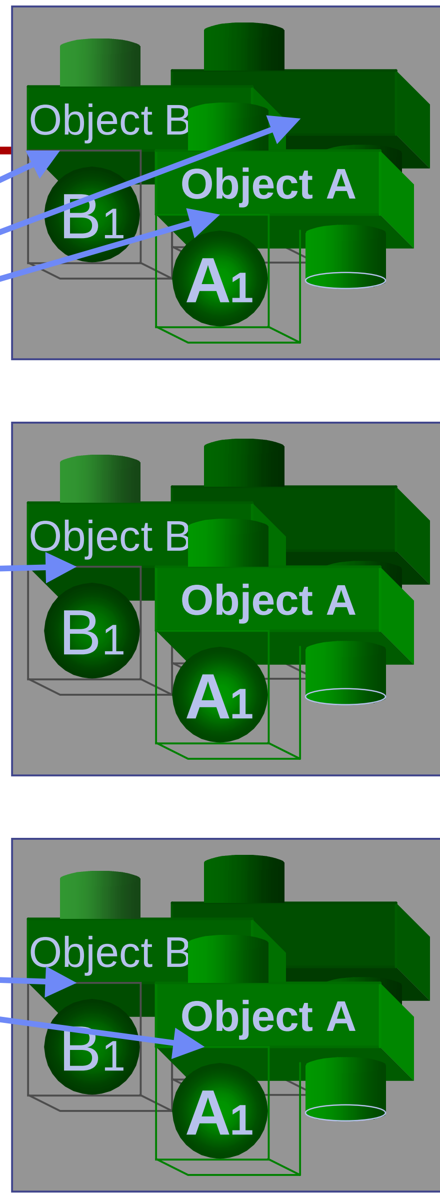
- Infrastructure-Controlled
  - Fault Tolerance Infrastructure maintains strong replica consistency of the object replicas using logging, checkpointing, activation, and recovery
- Application-Controlled
  - The application is responsible for maintaining whatever consistency it requires, using its own mechanisms
  - No logging, checkpointing, activation or recovery are provided by the Fault Tolerance Infrastructure

Strong Replica Consistency

- Maintained for object groups that have the Infrastructure-controlled Consistency Style
- For Active replication, at the end of each operation, all of the members of the object group have the same state
- For Passive replication, at the end of each state transfer, all of the members of the object group have the same state

Fault Monitoring Granularity

- Member
- Location (proxy object for the location)
- Location and Type (proxy object of given type for the location)



Factories

- Sequence of **FactoryInfo**
  - **Factory** that can be used to create a member of the object group
  - **Location** at which factory is to create a member of the object group
  - **Criteria** that the factory is to use when creating the member of the object group, e.g. initialization values, constraints on the member, etc

Break

Download specification (chapter 25 of CORBA 2.5) from

ftp://ftp.omg.org/pub/docs/formal/01-09-29.pdf

Download tutorial from

http://www.eternal-systems.com

http://www.vertel.com

Contact us at

priya@cs.cmu.edu

joey-garon@vertel.com

Outline

1. Introduction to Fault Tolerance
2. Fault Tolerance mechanisms
3. Fault Tolerance properties

   Break

4. Fault Tolerance management
5. Fault Tolerance applications
6. Fault tolerant hello server example

Outline

4. **Fault Tolerance Management**
   - a. **Replication Management**
   - b. Fault Management
   - c. Logging and Recovery Management

Replication Management

- Replication Manager maintains object groups (replicated objects) and fault tolerance properties of the object groups
  - Replication Style
  - Membership Style
  - Consistency Style
  - etc

Replication Management

- **Replication Manager** interface provides methods to register and obtain Fault Notifier
  - **register_fault_notifier()**
  - **get_fault_notifier()**
- **Replication Manager** interface inherits from
  - **Property Manager**
  - **Object Group Manager**
  - **Generic Factory**

Property Manager

- Fault tolerance properties may be defined
  - For all replicated objects (object groups)
  - For all replicated objects of a type
  - For a specific replicated object at creation
  - For executing replicated objects
- More specific definitions override more general definitions

Property Manager ***Interface***

- **set_default_properties()**
- **get_default_properties()**
- **remove_default_properties()**
- **set_type_properties()**
- **get_type_properties()**
- **remove_type_properties()**
- **set_properties_dynamically()**
- **get_properties()**

Property Manager ***Interface***

```
void set_type_properties(
   in TypeId type_id,
   in Properties overrides)
raises(InvalidProperty, UnsupportedProperty);

Properties get_type_properties(
   in TypeId type_id);

void remove_type_properties(
   in TypeId type_id,
   in Properties props)
raises(InvalidProperty, UnsupportedProperty);
```

When Can Properties Be Set?

	Default	Type	Creation	Dynamic
Replication Style	✔	✔	✔	
Membership Style	✔	✔	✔	
Consistency Style	✔	✔		
Fault Monitoring Style	✔	✔		
Fault Monitoring Granularity	✔	✔	✔	✔
Factories		✔	✔	✔
Initial Number of Replicas	✔	✔	✔	
Minimum Number of Replicas	✔	✔	✔	✔
Fault Monitoring Interval and Timeout	✔	✔	✔	✔
Checkpoint Interval	✔	✔	✔	✔

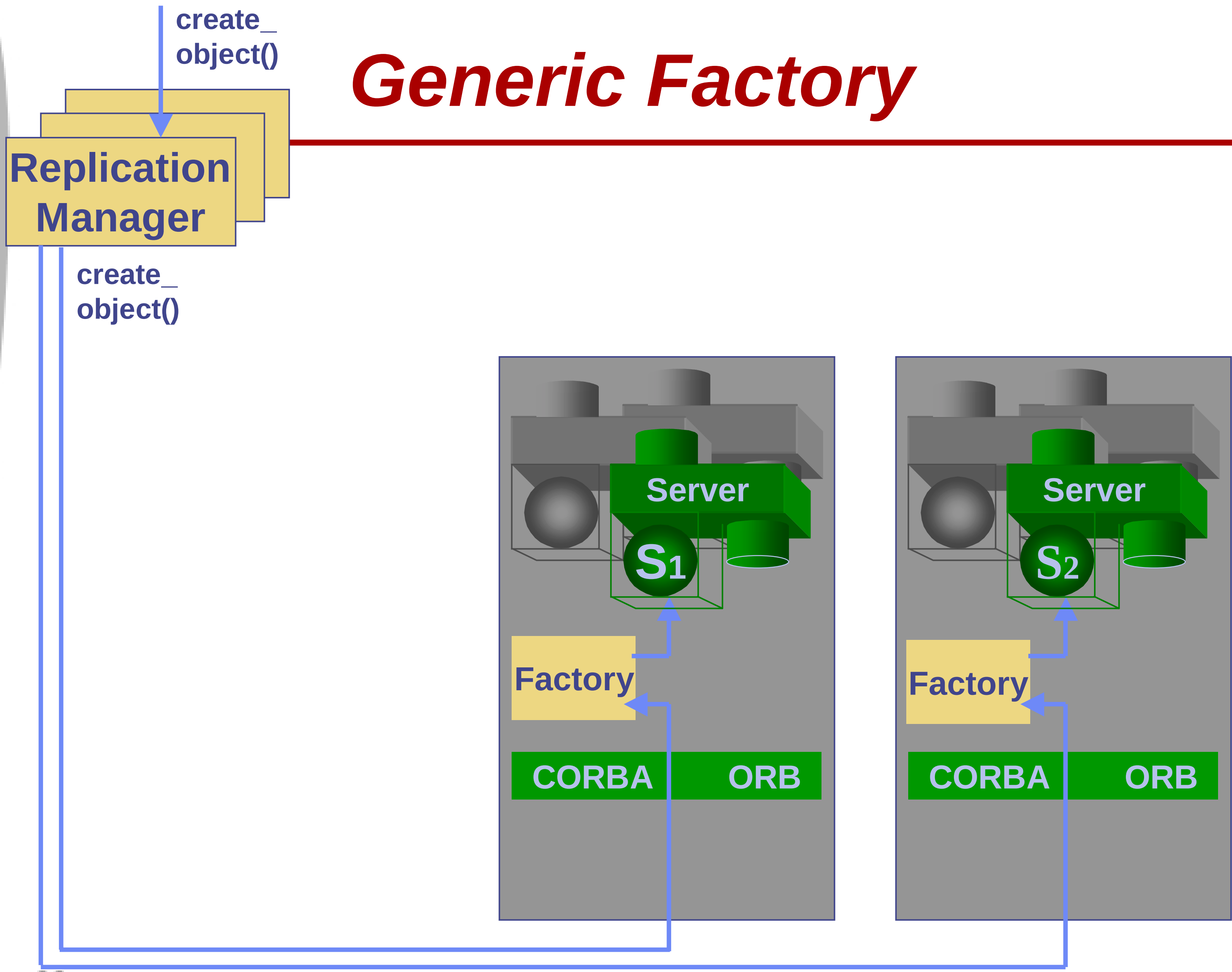
Generic Factory ***Interface***

- Inherited by Replication Manager and invoked by application to create or delete an object group
- Implemented by Application and invoked by Replication Manager or Application to create or delete an individual object replica
- **create_object()**
- **delete_object()**

Generic Factory ***Interface***

```
typedef Object ObjectGroup;
typedef any FactoryCreationId;

Object create_object(
   in TypeId type_id,
   in Criteria the_criteria,
   out FactoryCreationId factory_creation_id)
raises(NoFactory, ObjectNotCreated, InvalidCriteria,
        InvalidProperty, CannotMeetCriteria);

void delete_object(
   in FactoryCreationId factory_creation_id)
raises(ObjectNotFound);
```

Generic Factory



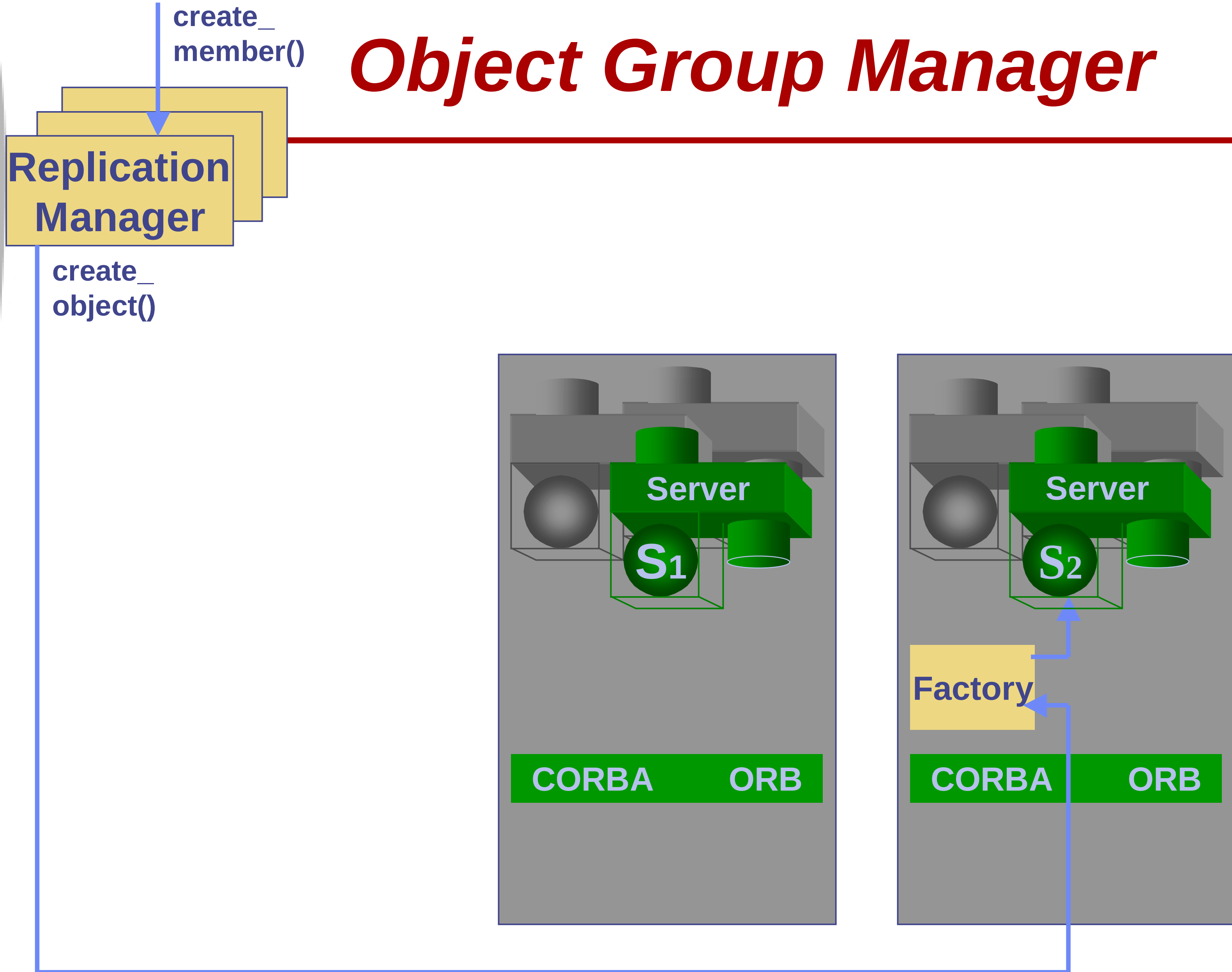
Object Group Manager *Interface*

- **create_member()**
- **add_member()**
- **remove_member()**
- **set_primary_member()**
- **locations_of_members()**
- **get_object_group_ref()**
- **get_object_group_id()**
- **get_member_ref()**

Object Group Manager ***Interface***

```
ObjectGroup create_member(
   in ObjectGroup object_group,
   in Location the_location,
   in TypeId type_id,
   in Criteria the_criteria)
raises(ObjectGroupNotFound, MemberAlreadyPresent,
        NoFactory, ObjectNotCreated, InvalidCriteria,...);

ObjectGroup add_member(
   in ObjectGroup object_group,
   in Location the_location,
   in Object member)
raises(ObjectGroupNotFound, MemberAlreadyPresent,
        ObjectNotAdded);
```

Object Group Manager



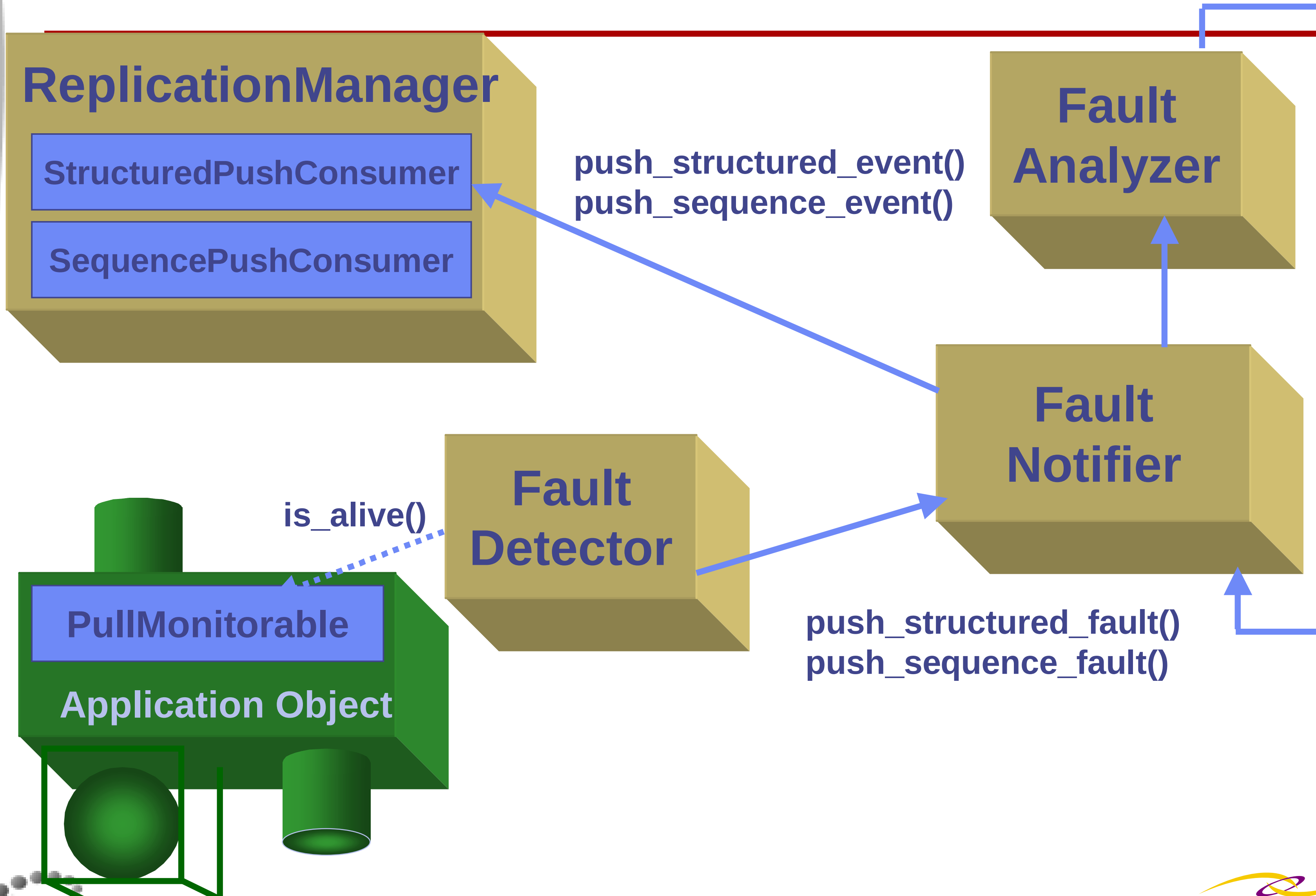
Outline

4. **Fault Tolerance Management**
   a. Replication Management
   b. Fault Management
   c. Logging and Recovery Management

Fault Management

- Fault Detector
  - Part of Infrastructure
  - Supplier of fault reports to FaultNotifier
- Fault Notifier
  - Receives fault reports from Fault Detectors and Fault Analyzer
- Fault Analyzer
  - Specific to Application
  - Both a consumer and a supplier of fault reports

Fault Detection & Notification



Fault Event Propagation

- Fault Event Propagation
  - CosNotification::StructuredEvent
  - CosNotification::EventBatch
- Types of Fault Event
  - ObjectCrashFault
- If all objects at a Location fail, TypeId and ObjectGroupId does not exist
- If all objects of a TypeId at a Location failed, ObjectGroupId does not exist

Domain_name = FT_CORBA	
Type_name = ObjectCrashFault	
FTDomainId	mydomain
Location	myhost/myprocess
TypeId	IDL:Bank:1.0
ObjectGroupId	1

Fault Event Suppliers & Consumers

- Fault Event Supplier
  - Fault Detector
  - Pushes fault events
- Fault Event Consumer
  - ReplicationManager, Consumer Object created by ReplicationManager, or Application
  - Registers using connect methods
  - Adds constraints to filter fault events propagated to it by the FaultNotifier

Fault Notifier ***Interface***

- Supplier End
  - push_sequence_fault()
  - push_structured_fault()
- Consumer End
  - connect_structured_fault_consumer()
  - connect_sequence_fault_consumer()
  - create_subscription_filter()
  - disconnect_consumer()

Fault Notifier ***Interface***

void **push_structured_fault**(
  in CosNotification::StructuredEvent event);

void **push_sequence_fault**(
  in CosNotification::EventBatch events);

Fault Notifier ***Interface***

typedef unsigned long long **ConsumerId**;

CosNotifyFilter::Filter **create_subscription_filter**(
  in string constraint_grammer)
  raises(CosNotifyFilter::InvalidGrammer);

ConsumerId **connect_structured_fault_consumer**(
  in CosNotifyComm::StructuredPushConsumer consumer,
  in CosNotifyFilter::Filter filter);

void **push_structured_fault**(
  in CosNotification::StructuredEvent event);

Outline

4. Fault Tolerance Management

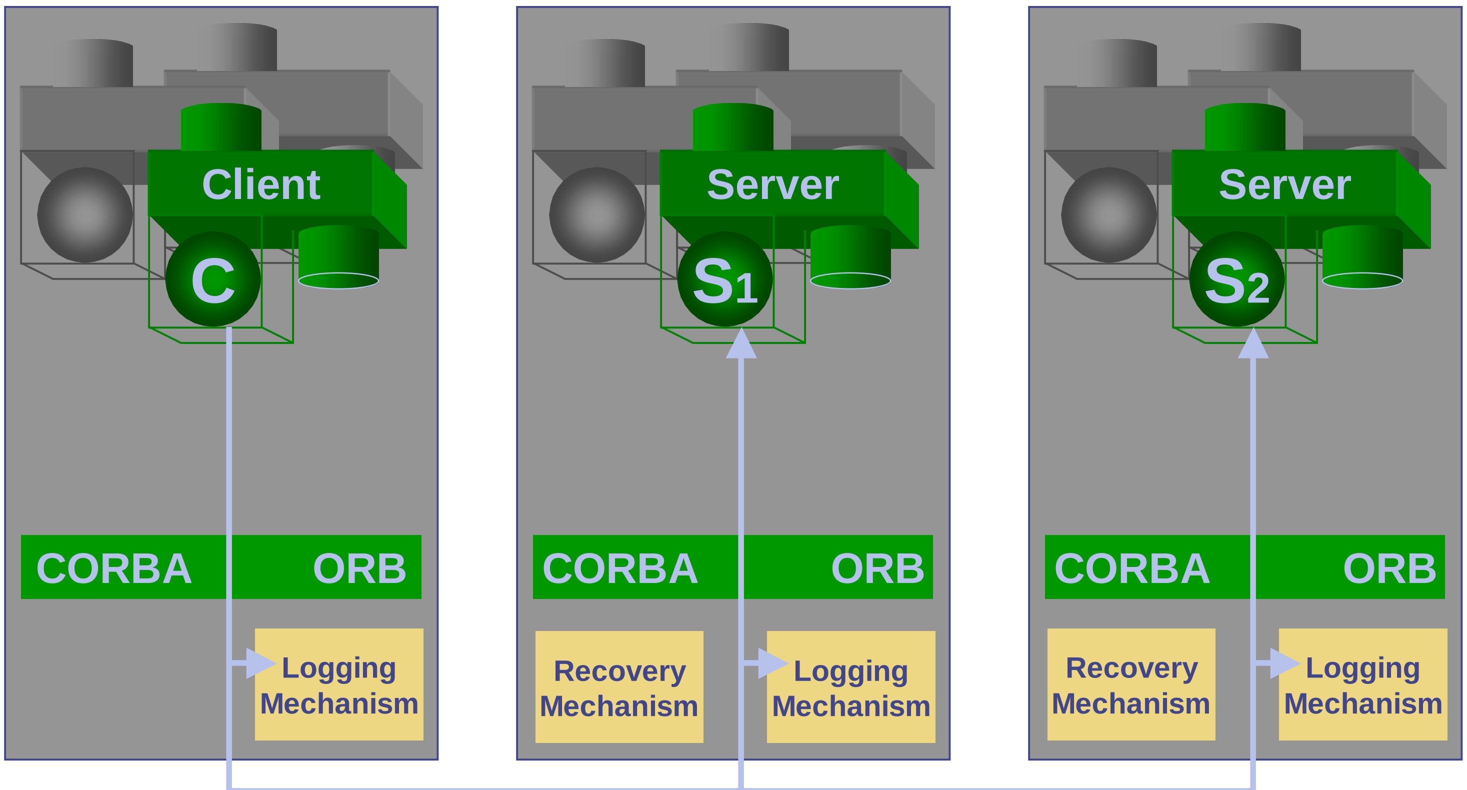
a. Replication Management

b. Fault Management

c. Logging and Recovery Management

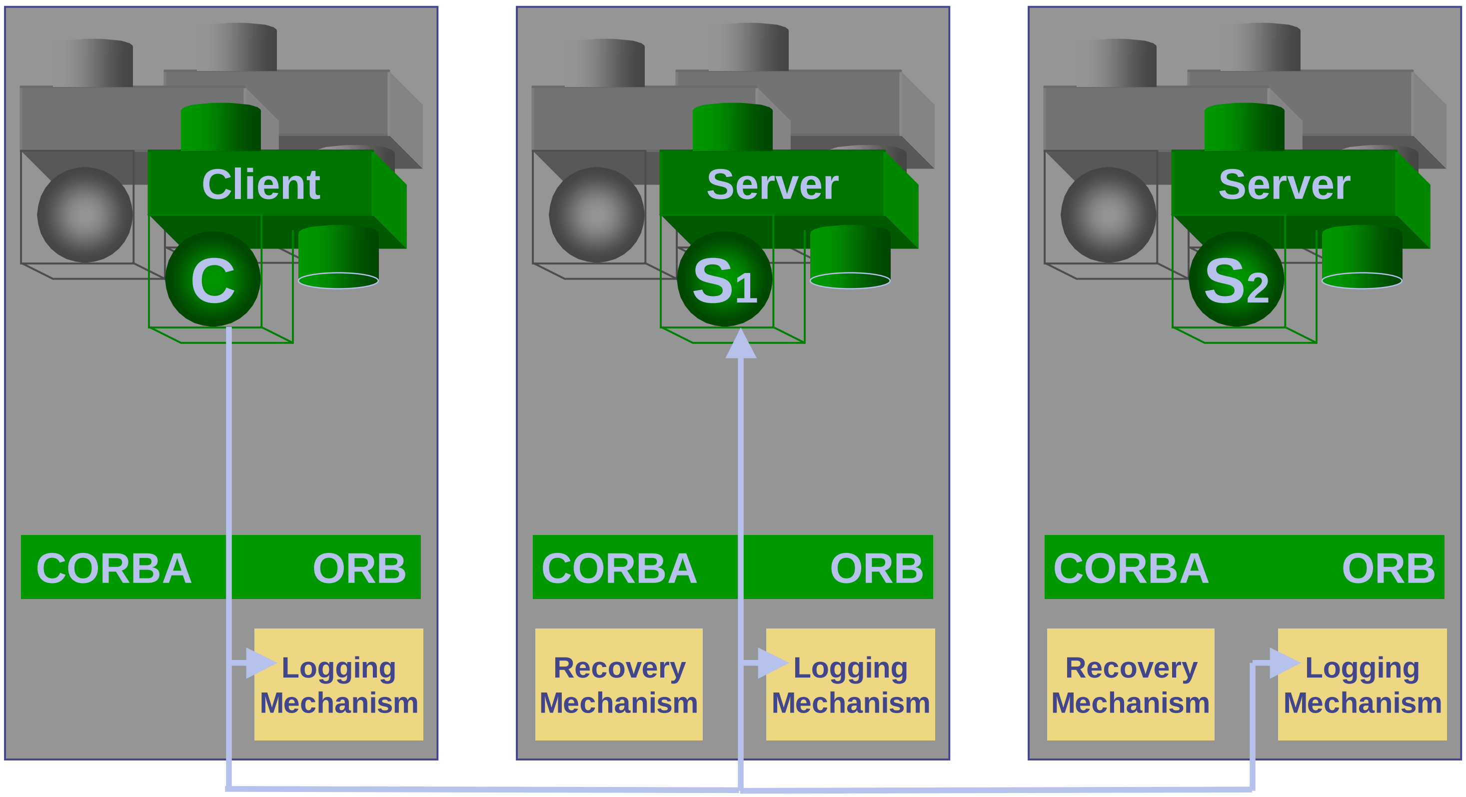
Logging & Recovery Management

Logging for Active Replication



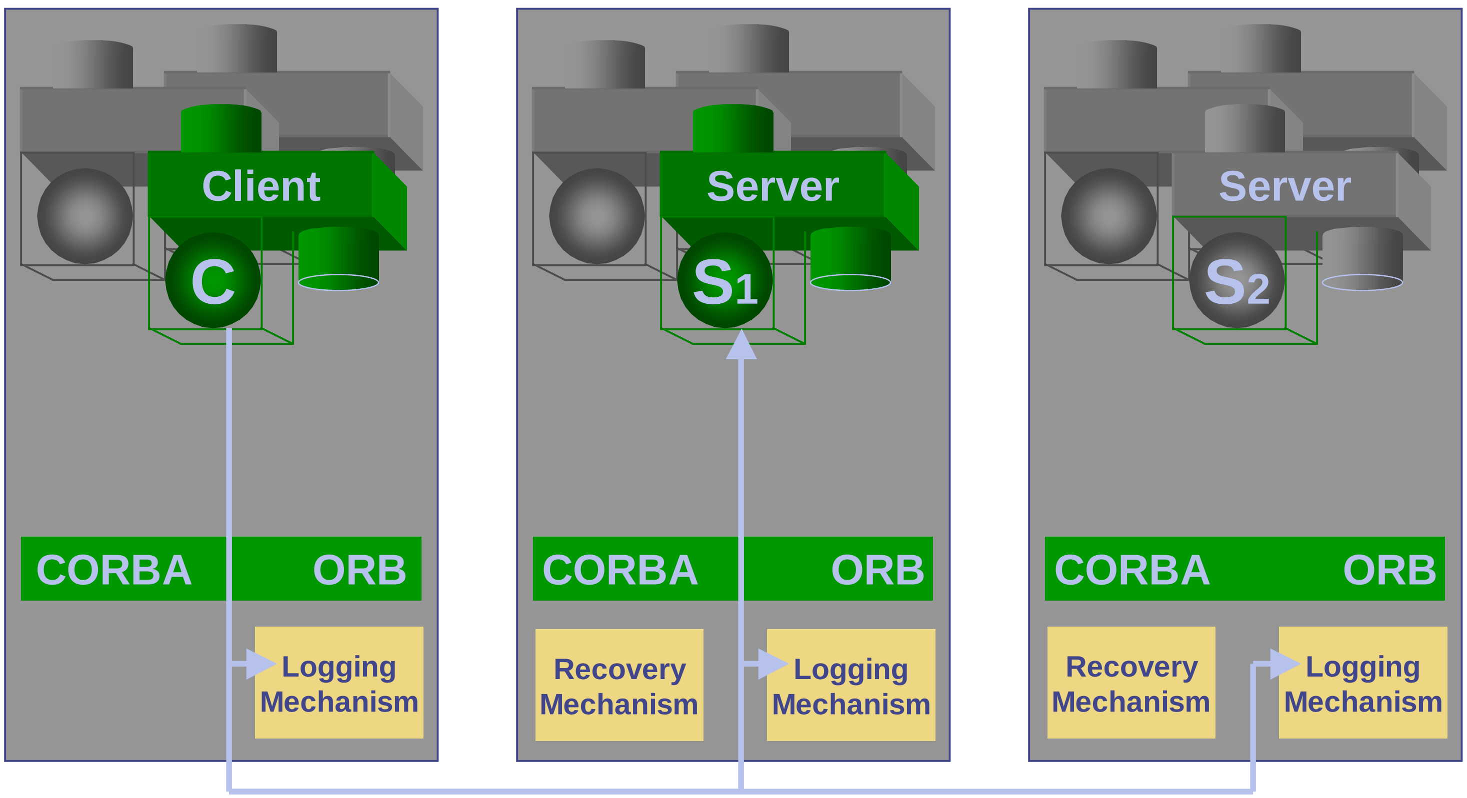
Logging & Recovery Management

Logging for Warm Passive Replication



Logging & Recovery Management

Logging for Cold Passive Replication



Checkpointable & Updateable *Interfaces*

Checkpointable *Interface*

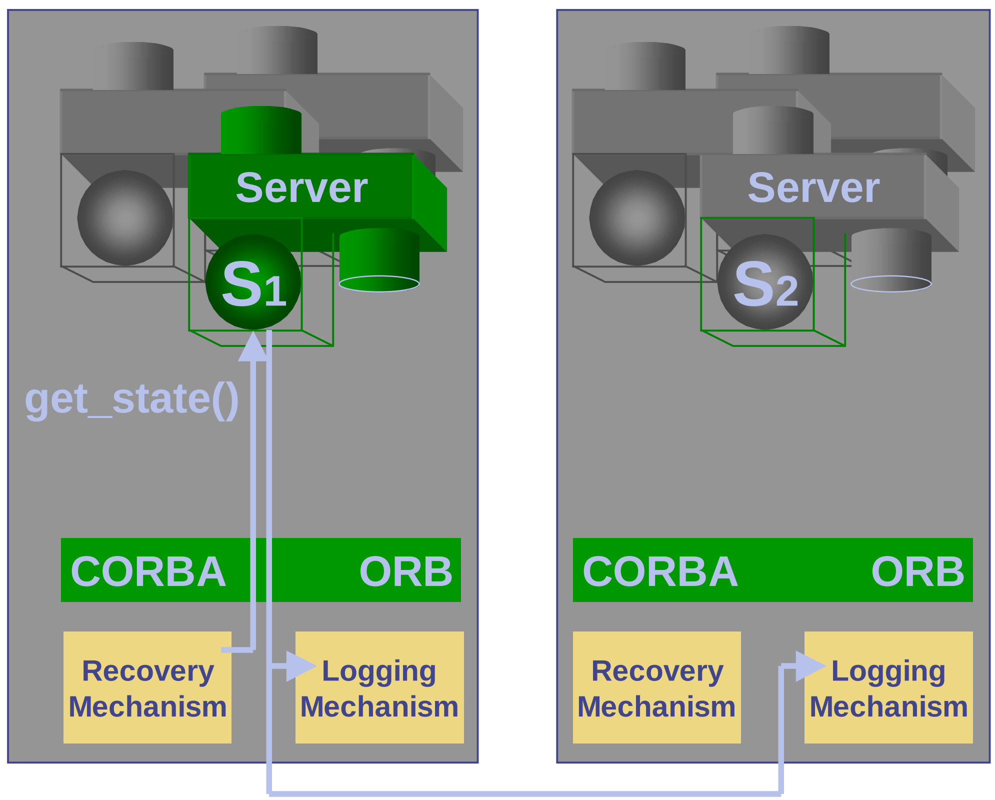
- **get_state()**
- **set_state()**

Updateable *Interface*

- **get_update()**
- **set_update()**

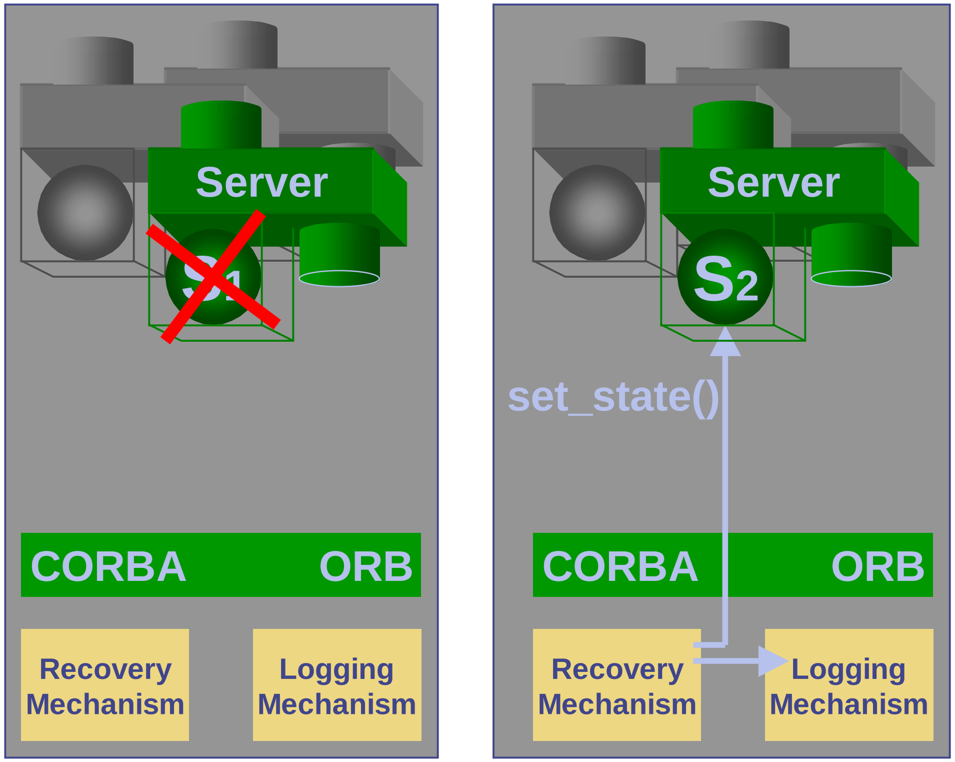
Logging & Recovery Management

State Transfer for Cold Passive Replication



Logging & Recovery Management

Recovery for Cold Passive Replication



Outline

1. Introduction to Fault Tolerance
2. Fault Tolerance mechanisms
3. Fault Tolerance properties

   Break

4. Fault Tolerance management
5. Fault Tolerance applications
6. Fault tolerant hello server example

Outline

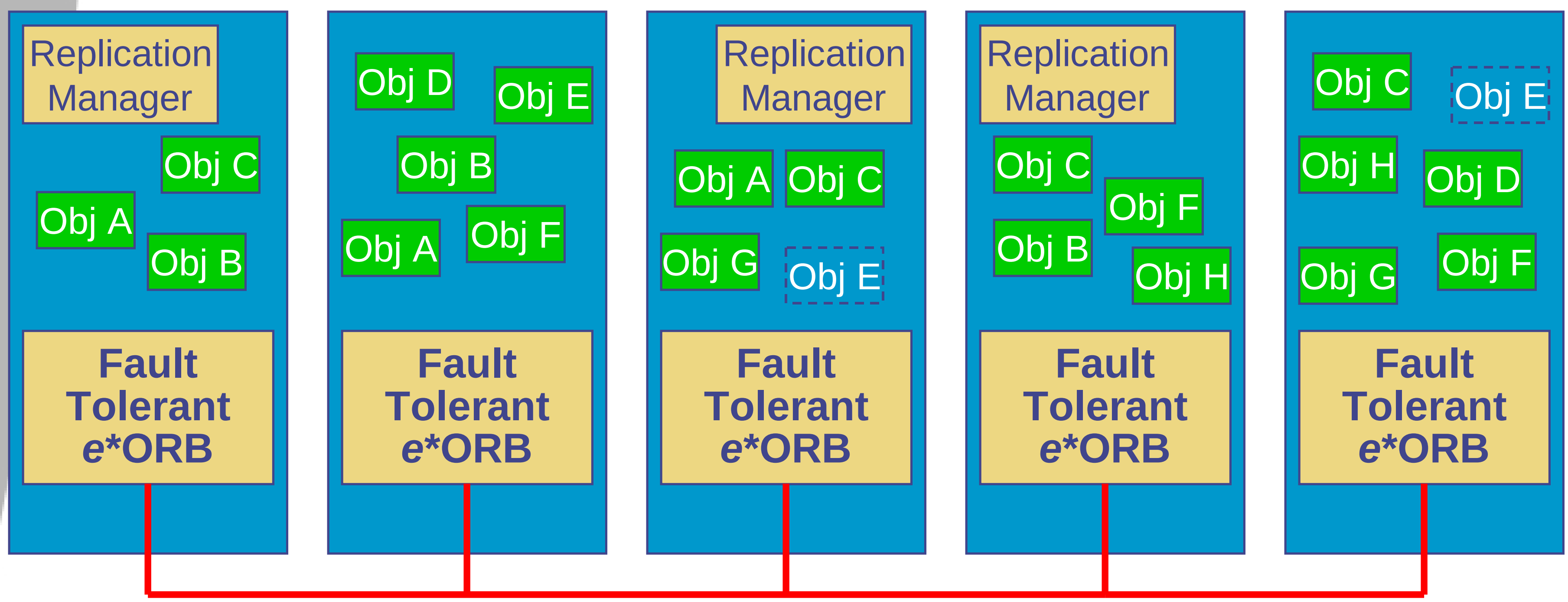
5. Fault Tolerant Applications
    a. Pool of Processors
    b. Internet Server
    c. Telco Switching

Pool of Processors

- Multiple replicas of each application object
- The replicas of an application object are assigned to different processors
- No need for all objects to have the same number of replicas, or the same type of replication
- Replication Manager is replicated just like any other object

Pool of Processors

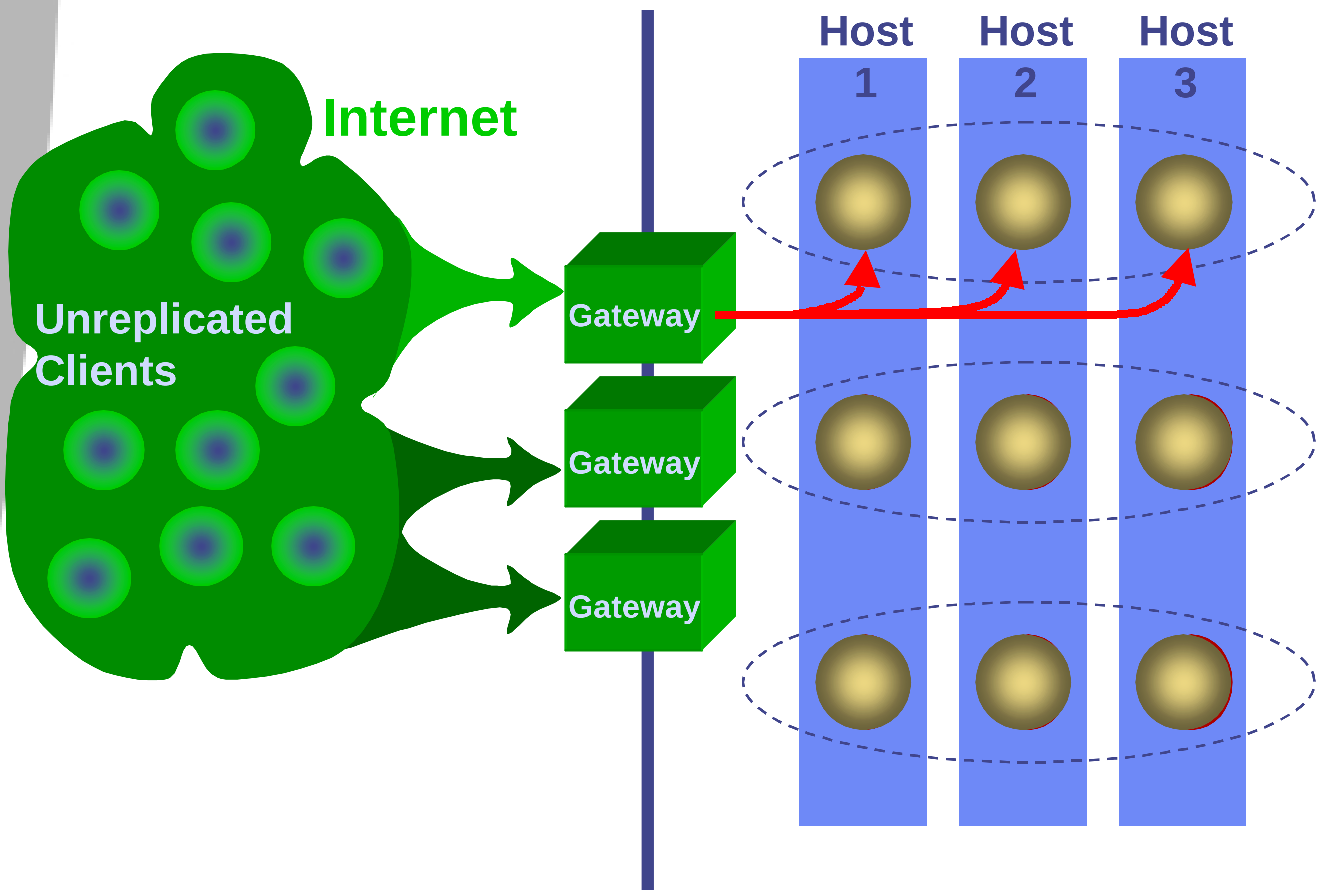
The replicas of an application object are assigned to different processors



Internet Server

- Use pool of processors
- Most clients will be outside our system and will not understand fault tolerance
- They communicate using IIOP/TCP/IP and enter the FT Domain through a gateway
- If a gateway fails, the clients can failover to another gateway

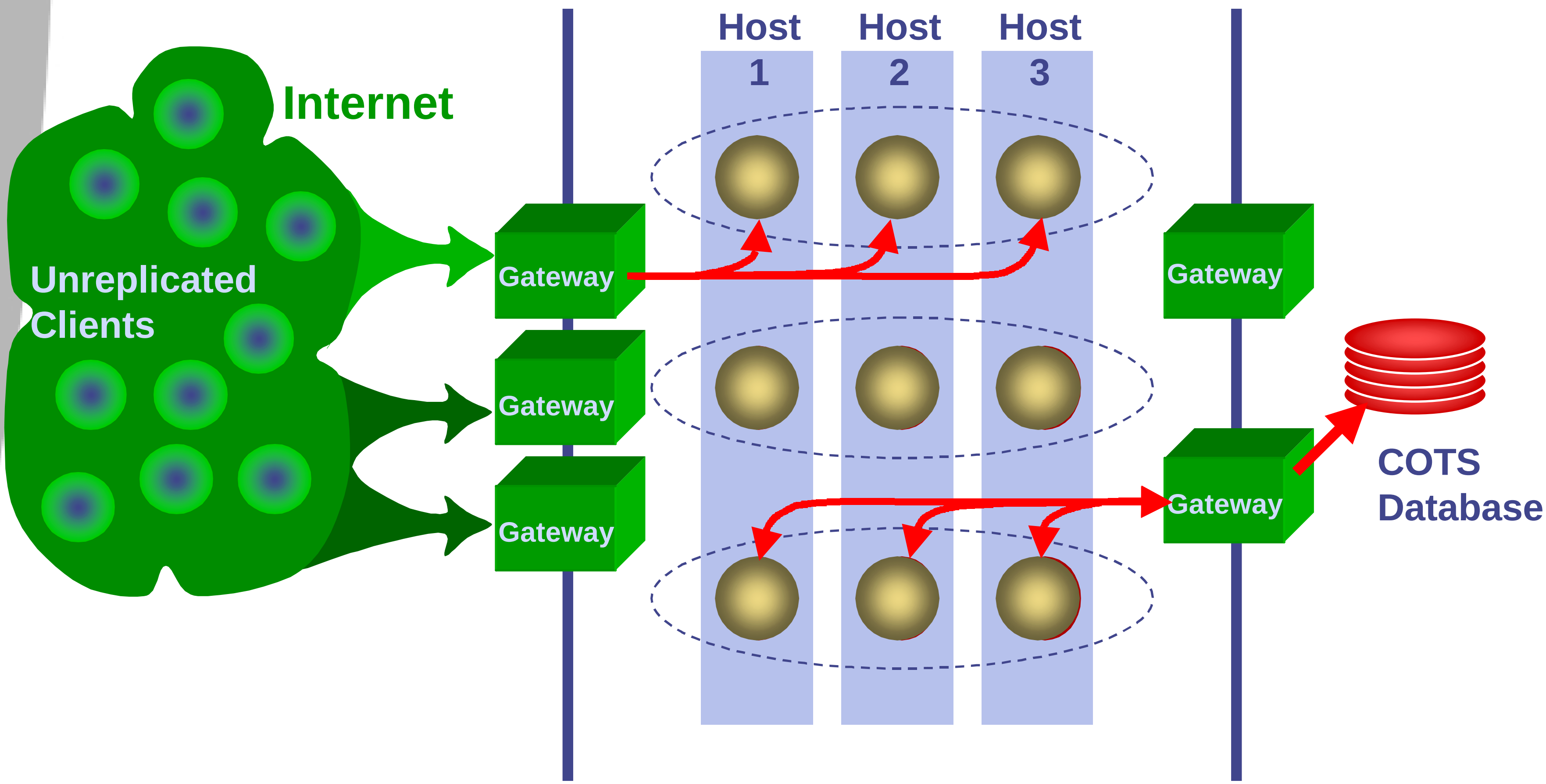
Internet Server



Internet Server

- Must also provide back-end database to record inventory, orders, etc.
- Do not attempt to replicate a database
- Use a COTS fault-tolerant database
- Access the database through a gateway
- The gateway ensures that
  - The database is accessed once only
  - Replies from the database are multicast to all replicas

Internet Server

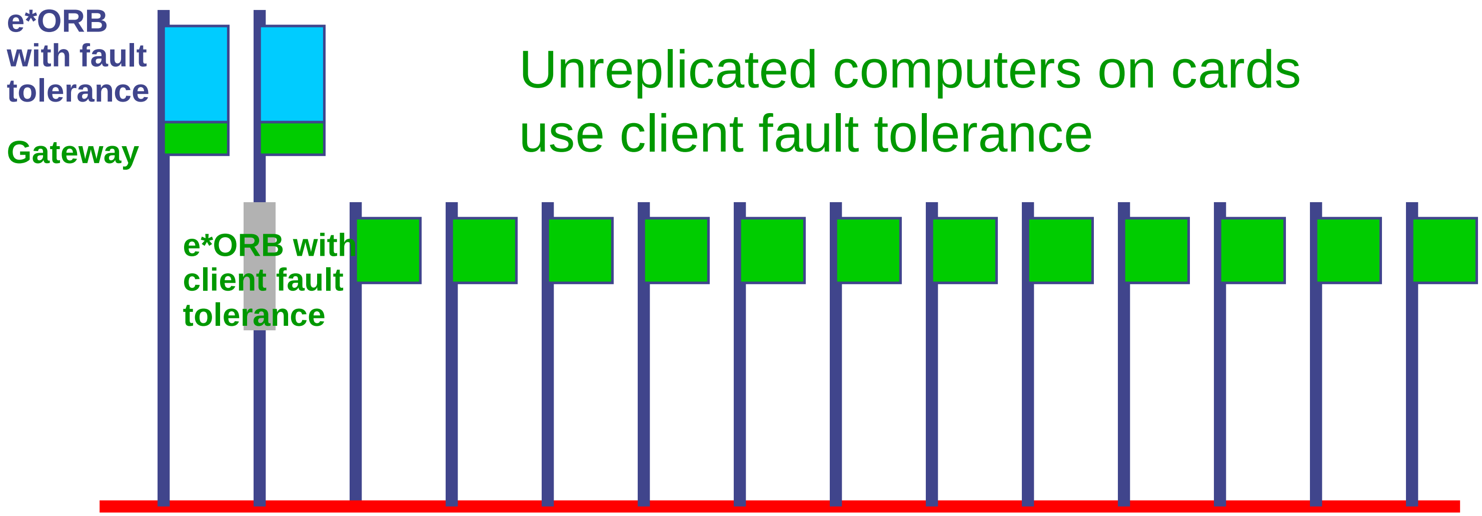


Simple Switching Application

- Line cards plugged into dual-bus backplane
  Each card has embedded processor with ORB
- Each line card is distinct; they are not replicas
- Two control processors use active replication
- Either control processor can control the switch
  They are true replicas
- Line cards communicate with both
  control processors

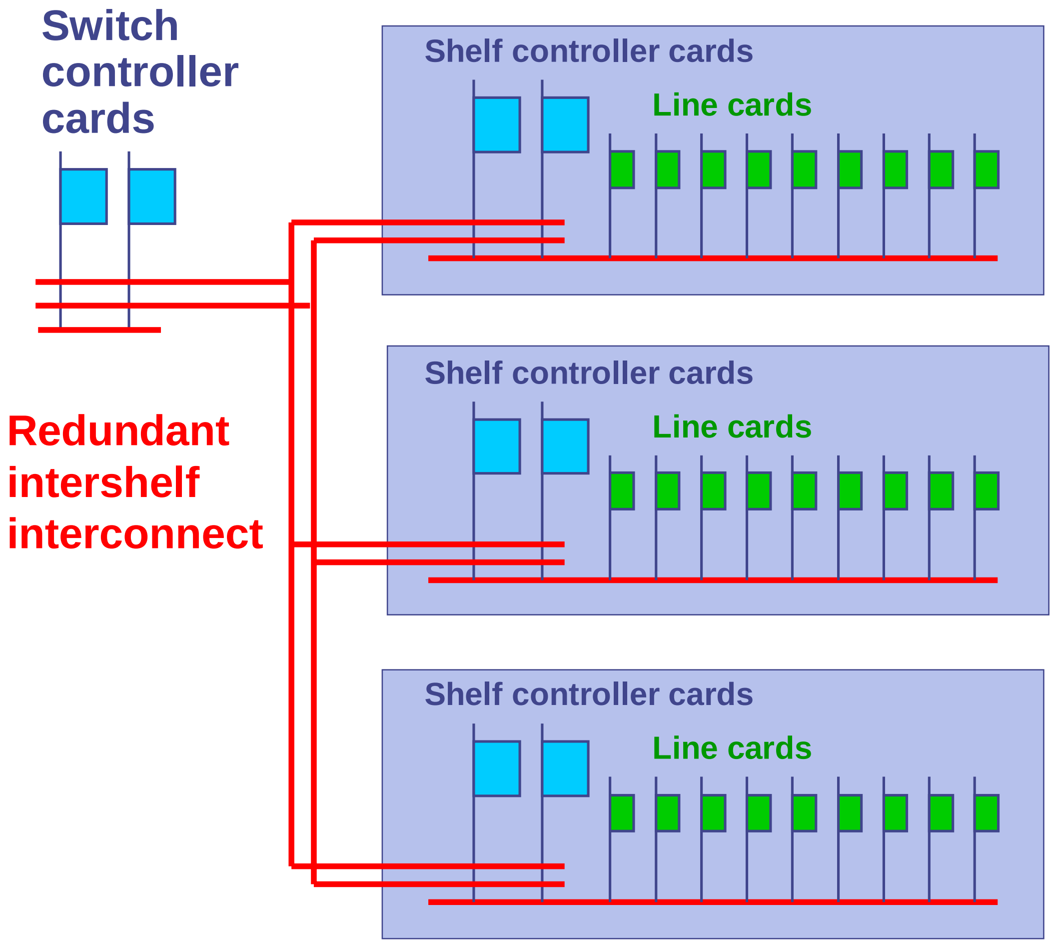
Simple Switching Application

Replicated Control Computers
use embedded fault tolerance
with active replication



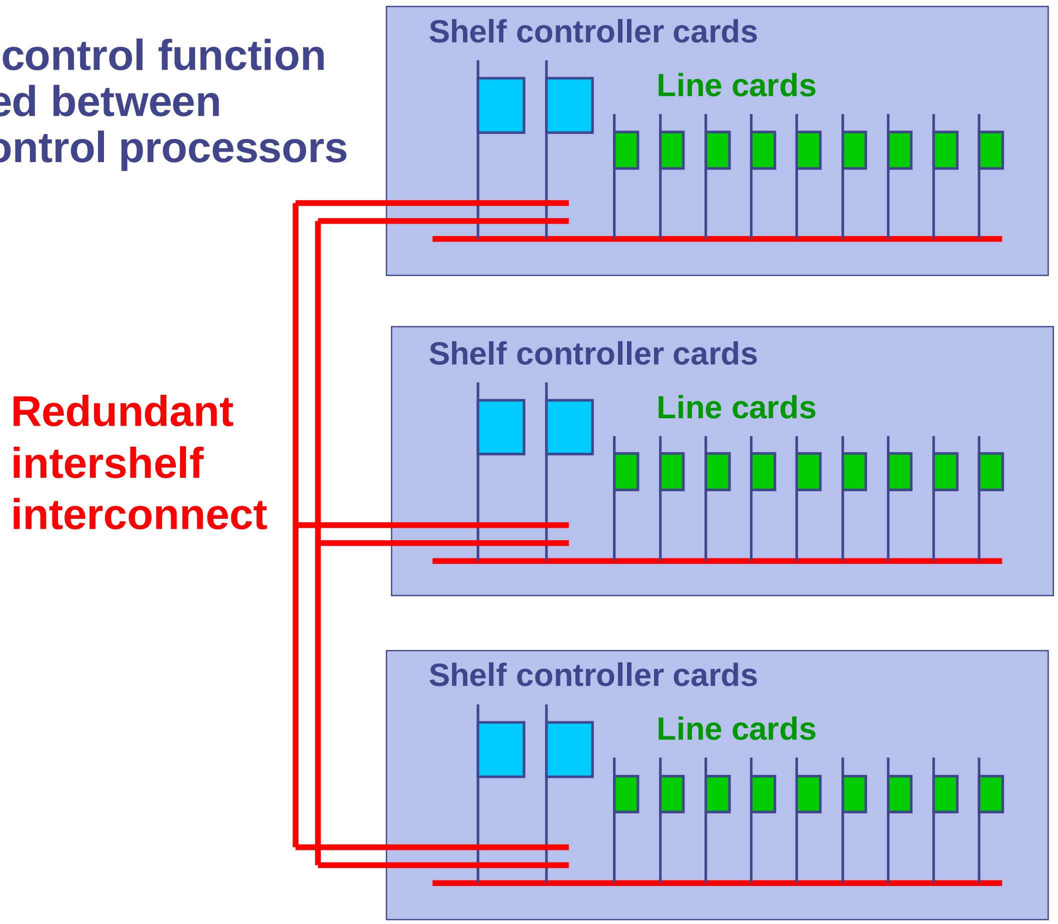
Backplane with dual bus interconnect

Larger Switching Application



Larger Switching Application

Switch control function is shared between shelf control processors



Outline

1. Introduction to Fault Tolerance
2. Fault Tolerance mechanisms
3. Fault Tolerance properties

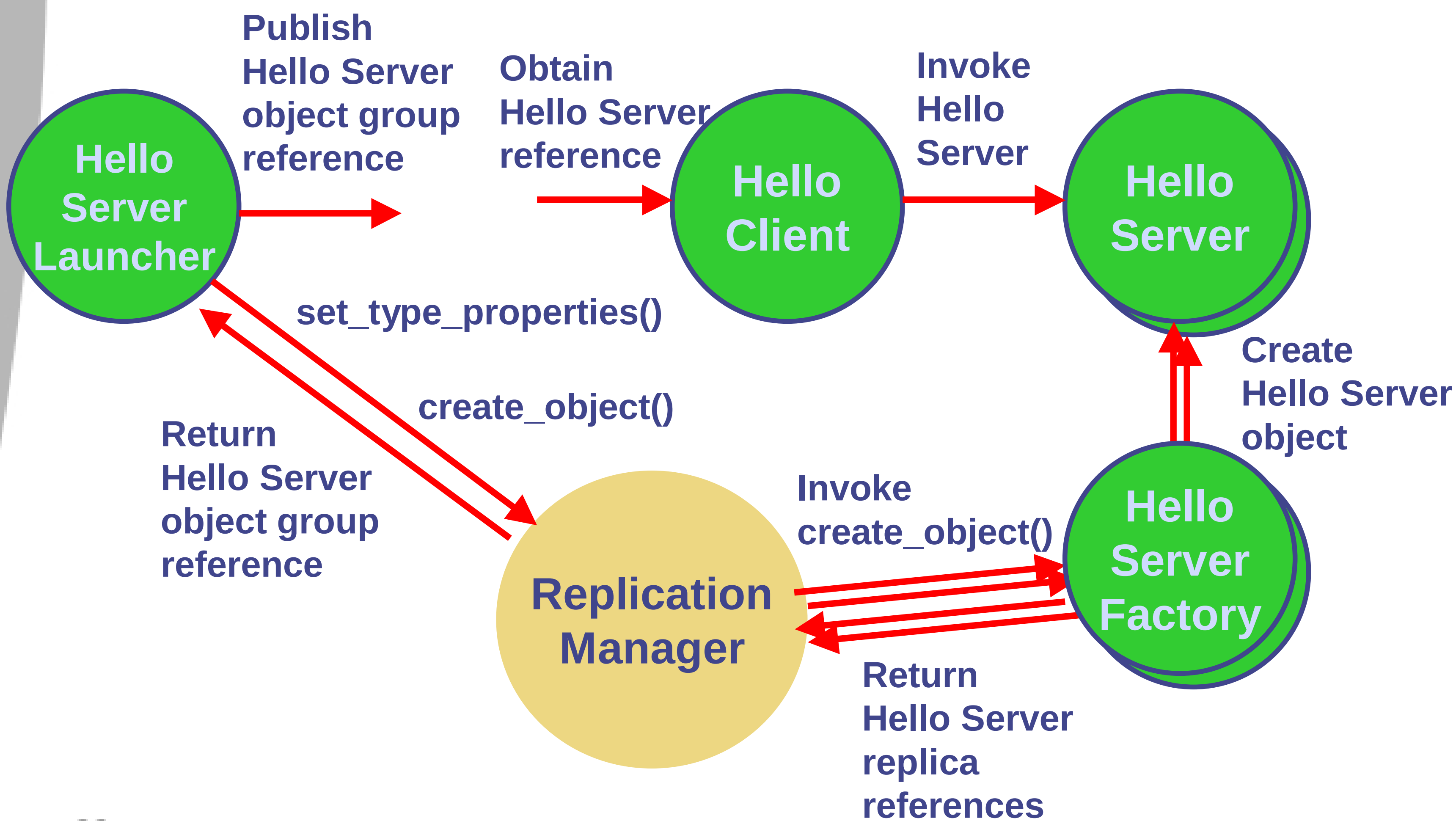
   Break

4. Fault Tolerance management
5. Fault Tolerance applications
6. Fault tolerant hello server example

Outline

6. **Fault-Tolerant Hello Server Example**
   a. **Hello Server Launcher**
   b. **Hello Server Factory**
   c. **Hello Server**
   d. **Hello Client**

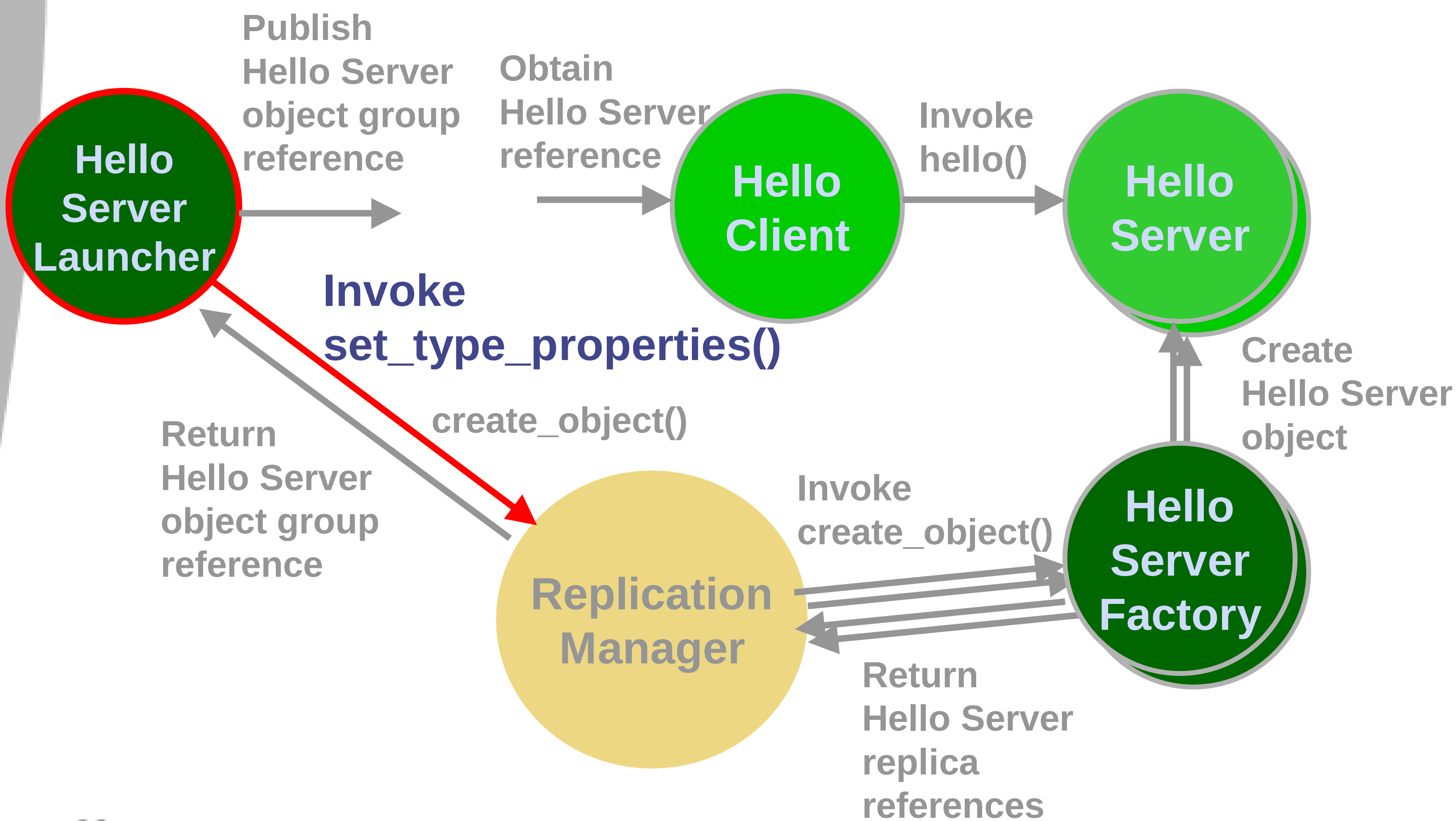
Hello Server Example



Hello Server Launcher

1. Initialize the ORB
2. Obtain a reference to the Replication Manager
3. Narrow the reference to the Property Manager
4. Invoke the set_type_properties() method of the Property Manager to set the properties for the Hello Server type
5. Narrow the reference to the Generic Factory
6. Invoke the create_object() method of the Generic Factory to create a Hello Server replicated object
7. Publish the Hello Server's IOGR in a file for the client to read

Hello Server Launcher



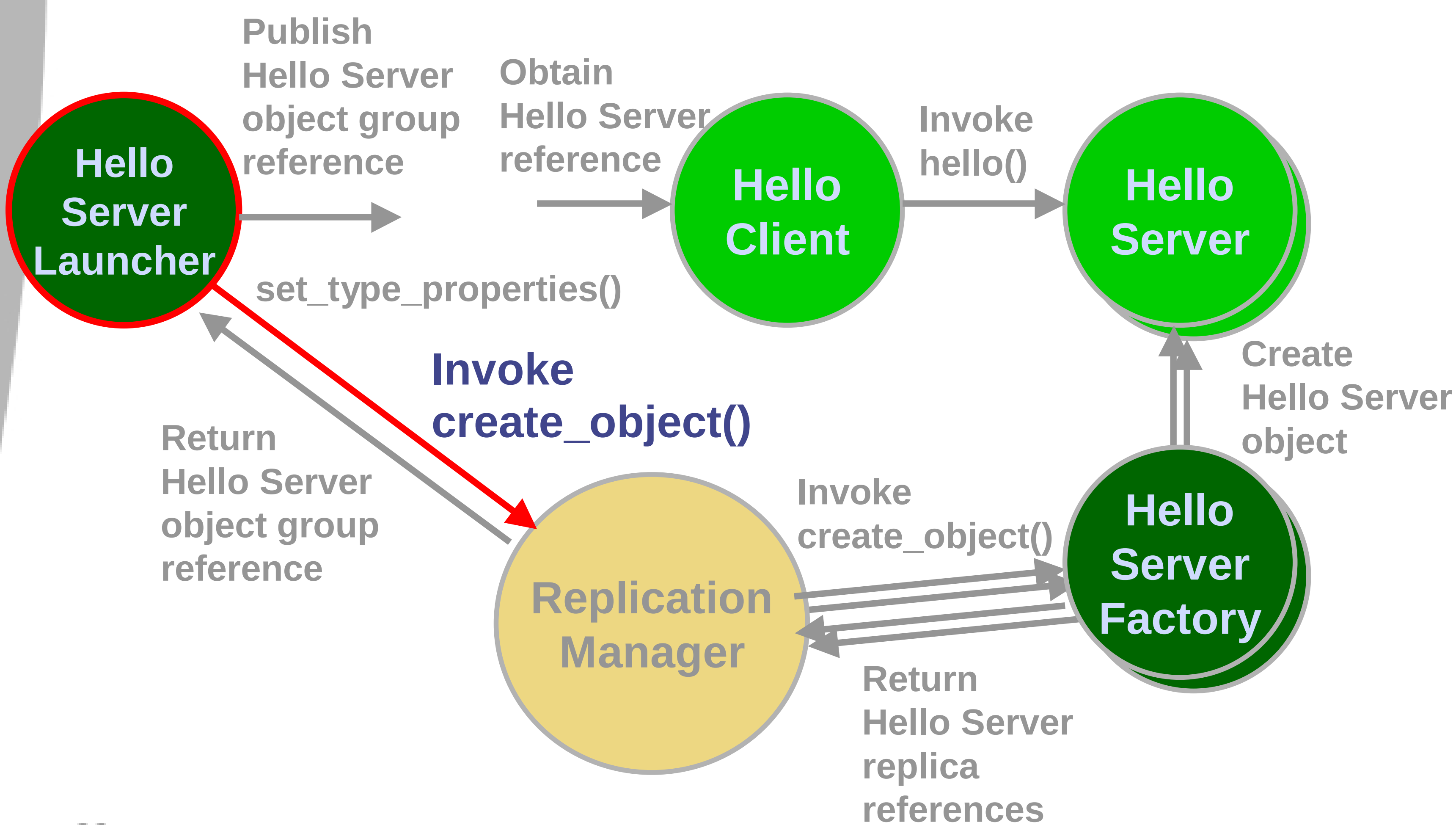
Hello Server Launcher Main

```
// Set type properties for the Hello Server type
try
{
  helloServertId = CORBA::string_dup("IDL:omg.org/HelloServer:1.0");
  helloServerProp.length(10);

  helloServerProp[0].nam.length(1);
  helloServerProp[0].nam[0].id
          = CORBA::string_dup("org.omg.ft.ReplicationStyle");
  helloServerProp[0].nam[0].kind = CORBA::string_dup("string");
  helloServerProp[0].val <<= FT::ACTIVE;
  ...
```

Hello Server Launcher Main

```
    helloServerProp[6].nam.length(1);
    helloServerProp[6].nam[0].id  =
          CORBA::string_dup("org.omg.ft.InitialNumberReplicas");
    helloServerProp[6].nam[0].kind = CORBA::string_dup("string");
    helloServerProp[6].val <<= (unsigned short)3;
      ...
    // Narrow the Replication Manager's object reference
    //  to a Property Manager reference and invoke
    //  set_type_properties()
    propMgr = FT::PropertyManager::_narrow(repMgr);
    propMgr->set_type_properties(helloServertId, helloServerProp);
}
//  Catch exceptions
```

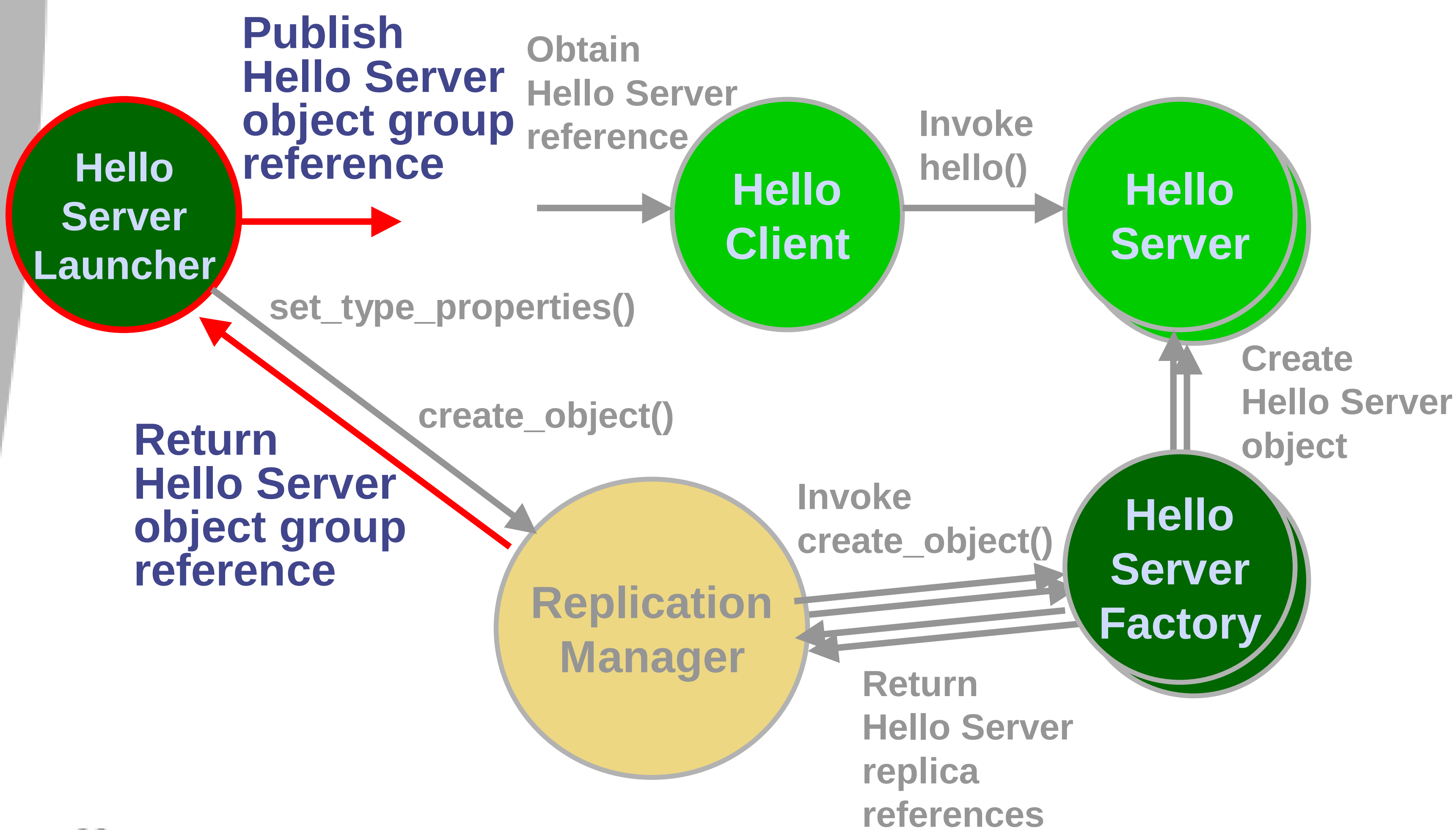
Hello Server Launcher



Hello Server Launcher Main

```
// Narrow the Replication Manager's object reference
// to a Generic Factory reference and invoke create_object()
genFact = FT::GenericFactory::_narrow(repMgr);
if (!CORBA::is_nil(genFact))
{
   theCriteria.length(0);
   helloServerRef = genFact->create_object(helloServerTId, theCriteria, fcId);
```

Hello Server Launcher



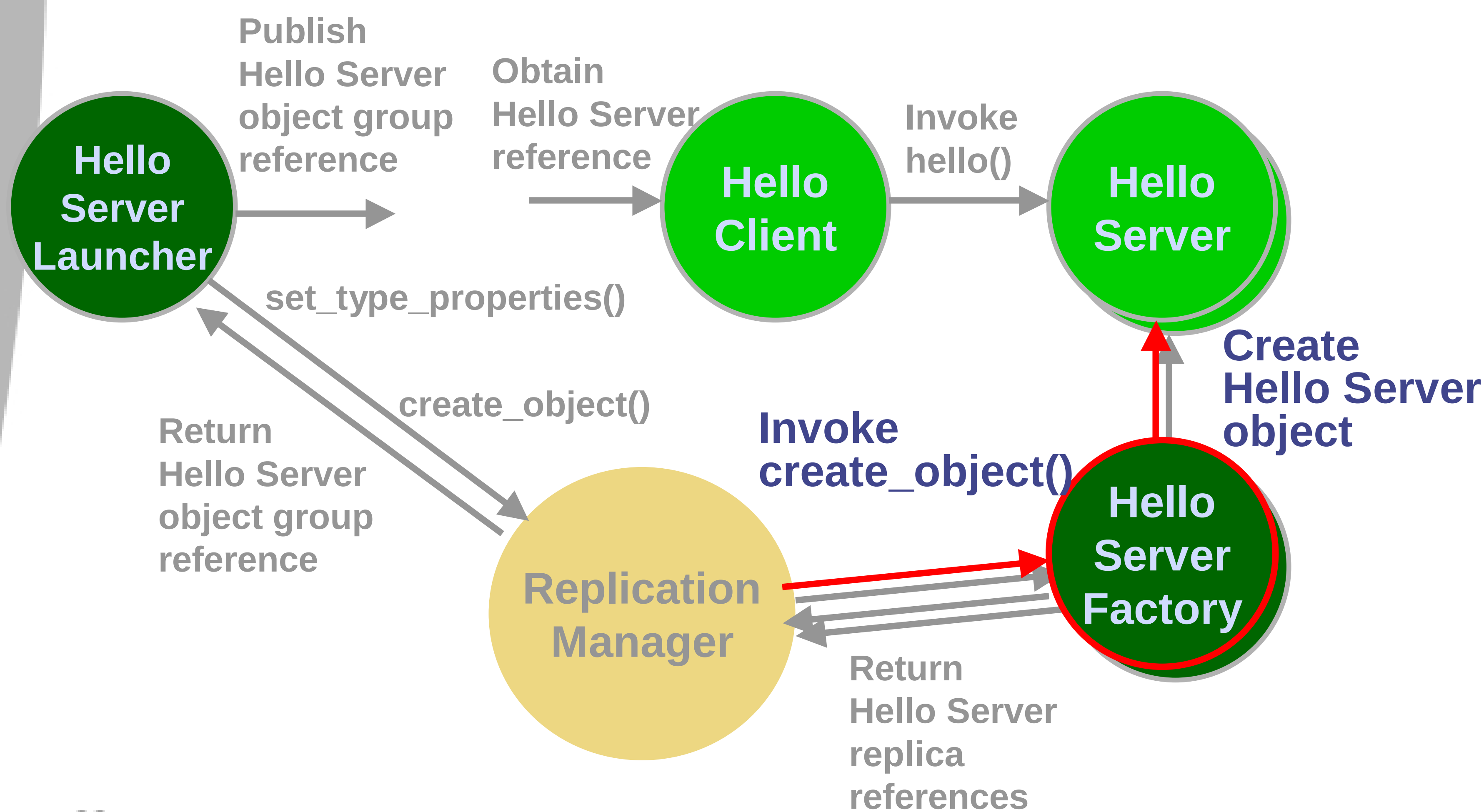
Hello Server Launcher Main

```
if (!CORBA::is_nil(helloServerRef))
{
  //  Publish the IOR of the Hello Server for the Client to read
  buffer = orb->object_to_string(helloServerRef);
  fd = fopen( "HelloServerIOR", "w" );
  if (fd == NULL)
  {  cerr << "Could not write file HelloServerIOR" << endl;
     exit(1);   }
  if (fputs(buffer, fd) == NULL)
  {  cerr << "Error in writing to file 'HelloServerIOR'" << endl;
     fclose(fd);
     exit(1);   }
  fclose(fd);
}
```

Hello Server Factory

1. **Invoked by FTCORBA rather than directly by the user**
2. **Extract the ObjectId from the criteria**
3. **Check the type_id to determine the object to create**
   - The factory may be able to create several types of objects
4. **Create the object and activate it**
5. **Record the object locally to enable deletion**
   - The index of the object in this local sequence is returned as an out parameter to enable deletion
   - A more sophisticated implementation of the factory would reuse indices
6. **Return the object reference of the object just created**

Hello Server Factory



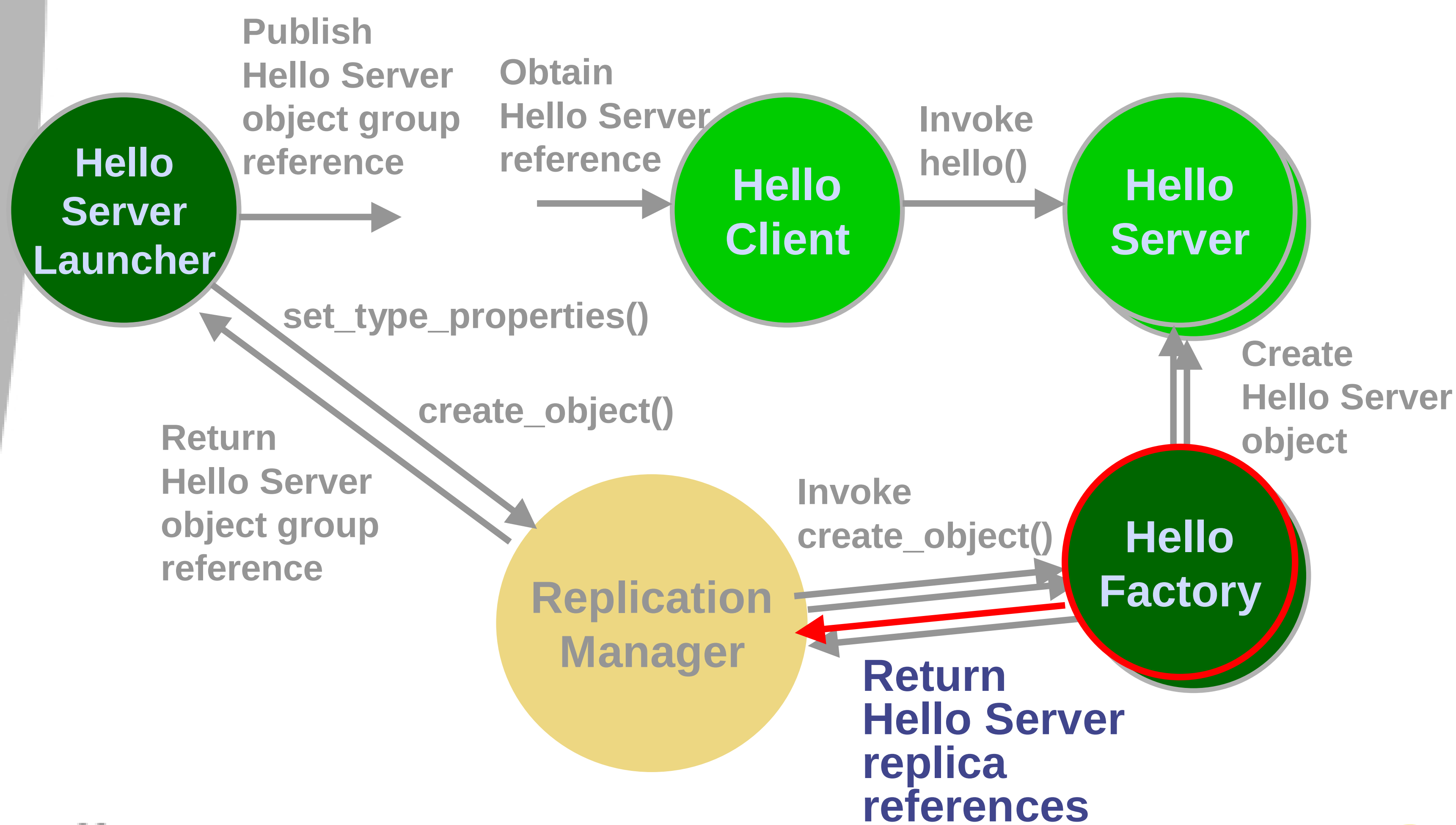
Hello Server Factory Implementation

```
CORBA::Object_ptr FactoryImpl::create_object(
                            const char* type_id, const FT::Criteria& the_criteria,
                            FT::GenericFactory::FactoryCreationId_out  factory_creation_id)
                      throw(FT::NoFactory, …)
 {  CORBA::Object_ptr helloServerRef;
    PortableServer::ObjectId *objId;
    int i, n, found;
    try
    {  i = 0;  found = 0;  n = the_criteria.length();
       while ((i<n) && (found==0))
       {  if ((the_criteria[i].nam.length() == 1) &&
                (strcmp(the_criteria[i].nam[0].id, "OBJECTID") == 0) &&
                (strcmp(the_criteria[i].nam[0].kind, "string") == 0))
          {  found = 1;
             assert(the_criteria[i].val >>= objId);
          }
          i++;
       }
       if (found == 0) { …
```

Hello Server Factory Implementation

```
if (strcmp(type_id, "IDL:omg.org/HelloServer:1.0") == 0)
{
   helloServerServant = new HelloServerImpl();
   myPoa->activate_object_with_id(*objectId, helloServerServant);
   helloServerRef = myPoa->create_reference_with_id(
                                                    *objectId,  “IDL:omg.org/HelloServer:1.0");
}
else
{
   throw FT::NoFactory();
}
```

Hello Server Factory



Hello Server Factory Implementation

```
  //  create_object() continued

  numberOfObjects++;
  objectSeq.length(numberOfObjects);
  objectSeq[numberOfObjects-1] = CORBA::Object::_duplicate(helloServerRef);

  factory_creation_id = new FT::GenericFactory::FactoryCreationId();
  (*factory_creation_id) <<= numberOfObjects;

  return helloServerRef;
 }
 catch(...)
 {
  throw FT::ObjectNotCreated();
 }
}
```

Hello Server Factory Main

1. **Initialize the ORB**
2. **Initialize the POA**
3. **Create the Factory object**
   - The Factory object is only invoked locally by FTCORBA
     Its object reference must never escape from this process
4. **Initialize FTCORBA**
   - Connects to the Replication Manager
   - Receives commands to create objects and
     invokes the Factory to create the local replica
5. **Set the ORB running**

Hello Server Factory Main

```
int main(int argc, char** argv)
{
  try
  {
    myOrb = CORBA::ORB_init(argc, argv);

    if (!CORBA::is_nil(myOrb))
    {
      rpObj = myOrb->resolve_initial_references("RootPOA");
      myPoa = PortableServer::POA::_narrow(rpObj);
```

Hello Server Factory Main

```
if (!CORBA::is_nil(myPoa))
{
   // Create myFactory servant
   myFactoryServant = new FactoryImpl();
   myFactoryOid = myPoa->activate_object(myFactoryServant);

   // Create an object reference for myFactory servant
   tempMyFactory = myPoa->create_reference_with_id(
                        *myFactoryOid, "IDL:omg.org/Factory:1.0");

   // Narrow the reference to the Generic Factory interface
   myFactory = FT::GenericFactory::_narrow(tempMyFactory);
```

Hello Server Factory Main

```
// Must NOT export the object reference for myFactoryServant
// New replicated objects are created by invoking the
// create_object() method of the Replication Manager

// Initialize FTCORBA
FTCORBA_init(myOrb, myPoa, myFactory, argc, argv);

// Using the ORB, make myFactory ready to receive requests
myOrb->run();
}
```

Hello Server Implementation

The implementation of Hello Server is very simple

1. **Obtain the name of the client as a parameter**
2. **Append “Hello” to the front of it**
3. **Append “!” to the back of it**
4. **Return the reply**

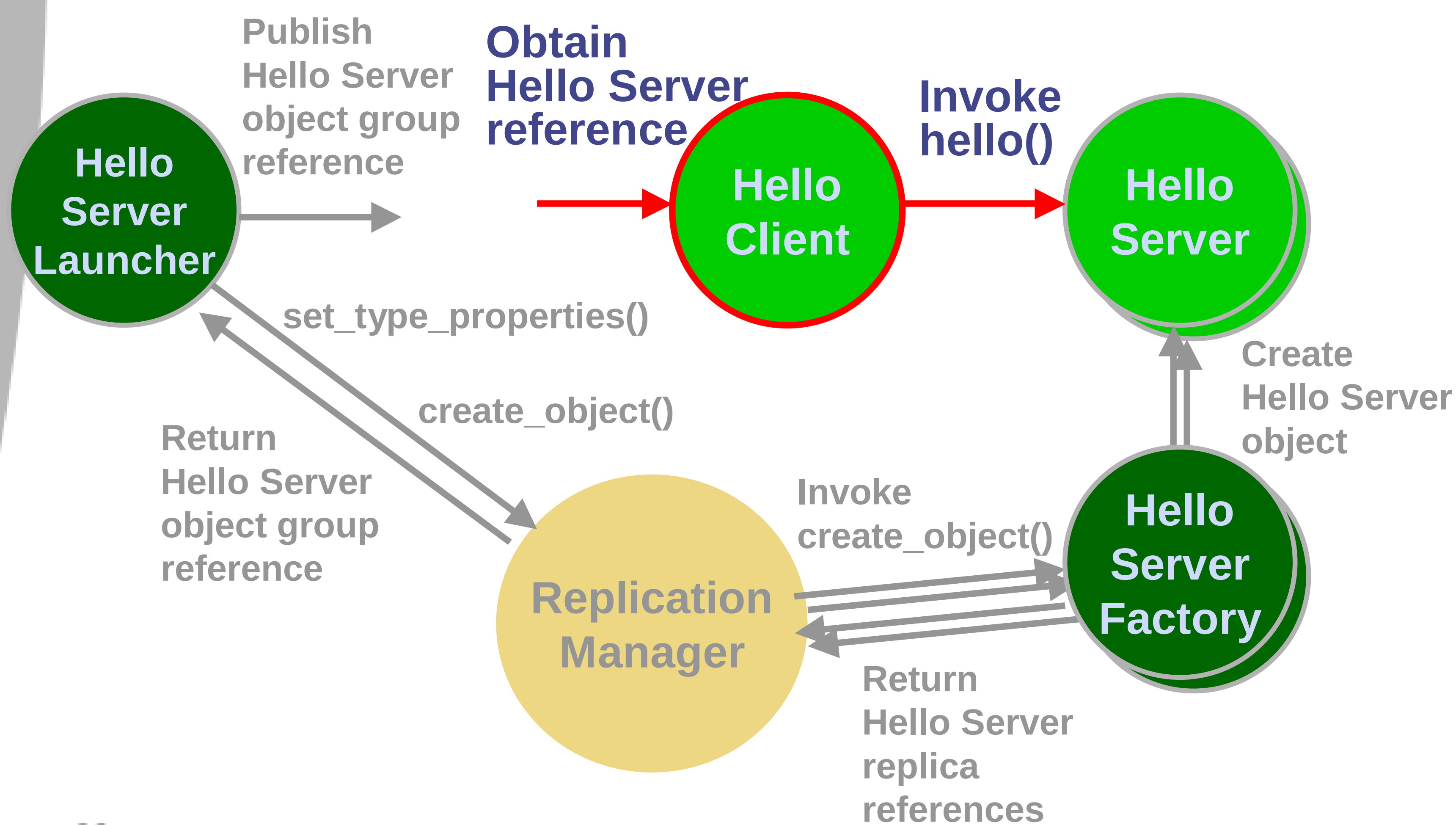
Hello Server Implementation

```
CORBA::String HelloServerImpl::hello(const char* hellostring)
{
  char hellostring2[200];

  strcpy(hellostring2, "Hello ");
  strcat(hellostring2, hellostring);
  strcat(hellostring2, "!");

  return CORBA::string_dup(hellostring2);
}
```

Hello Client Main

1. Initialize the ORB
2. Obtain a reference to an active Hello Server
3. Invoke the hello() method of the Hello Server

Hello Client



Hello Client Main

```
// Obtain the Hello Server Object Reference:  obj
…
// Narrow the object to a Hello Server
HelloServer_var server = HelloServer::_narrow(obj);

if (!CORBA::is_nil((HelloServer_ptr)server))
{
  CORBA::String_var returned;
  const char* hellostring = "client";

  //  Invoke the hello() method of the remote server
  returned = server->hello(hellostring);
  cout << returned << endl;
 }
```

Q & A

www.eternal-systems.com
www.vertel.com

Fault Tolerant CORBA

- For more information, contact us at:

priya@cs.cmu.edu

joey-garon@vertel.com

or Eternal Systems at +1-805-893-4897

Thank You!

www.eternal-systems.com
www.vertel.com