

---

# SRF Design Review

**Keys Botzum**

**[kbotzum@eos.hitc.com](mailto:kbotzum@eos.hitc.com)**

---

**17 April 1996**

# Server Request Framework (SRF) Overview



- **Driving Requirements**
- **SRF Issues Status**
- **SRF Context**
- **SRF Software Design**
  - high level pictures & capability description
  - object model
  - outline of class usage
  - event trace
  - detailed pseudo-code



# SRF Driving Requirements

## SRF Provides

- a framework for constructing ECS Servers and Client/Server APIs
- a common implementation of asynchronous request processing

## Evolutionary Features

- encapsulation of communication technology
  - eases transition to other technologies (e.g., CORBA)
  - allows multiple communication protocols

## Release B Status

- reusing Release A SRF design & implementation

See 305-CD-028-002 Section 4.5.2

# SRF Issues Status



## Infrastructure Review Concerns

- **performance**
  - **we don't expect significant overhead (discussed at Infrastructure Review)**
  - **will measure performance using SRF implementation**
- **layering cost vs. saved code**
  - **some performance cost, but we estimate 2-3 thousand lines of code saved per "typical" ECS server**
    - common dispatching/prioritizing infrastructure**
    - common recovery infrastructure**
    - common coding model**
- **CDS naming**
  - **SRF clients no longer register in CDS (change to CSS Message Passing layer)**

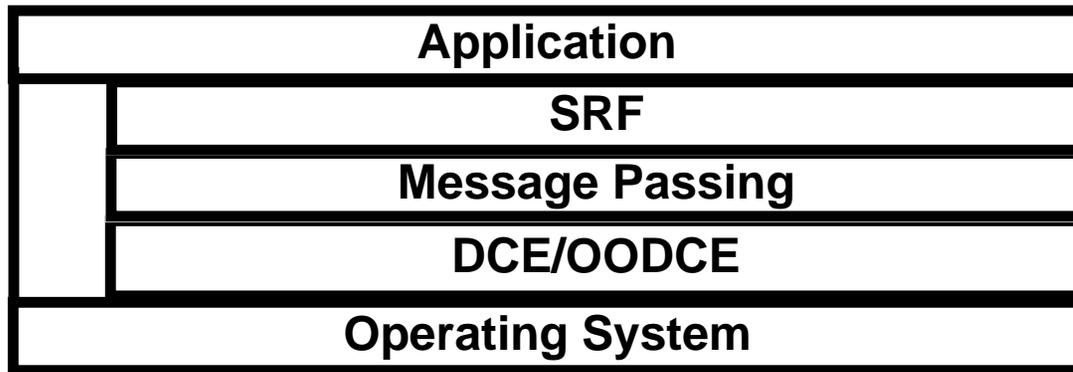
# SRF Context



- **SRF provides client & server functionality**
- **Clients will use SRF if the corresponding server provides an SRF interface**
- **Servers that will support SRF interfaces**
  - **Advertising Server**
  - **DIM/LIM Servers**
  - **Ingest**
  - **Gateway**



# SRF Stack



- **SRF provides an interface to the application developer**
- **builds on Message Passing layer & DCE/OODCE**



# SRF Capabilities

## Encapsulates common requirements

- for asynchronous request processing
- for authorization
- for priority based scheduling of requests

## Provides an extensible request dispatcher object for server construction

- base class implements priority based dispatching
- extensible for other dispatching policies (e.g., resource utilization)

## Provides Client & Server objects for request creation/submission

## Provides Bi-directional Client & Server objects for asynchronous request tracking and control

## Accessible through 5 common classes

- additional capabilities added through sub-types

## Hides implementation details

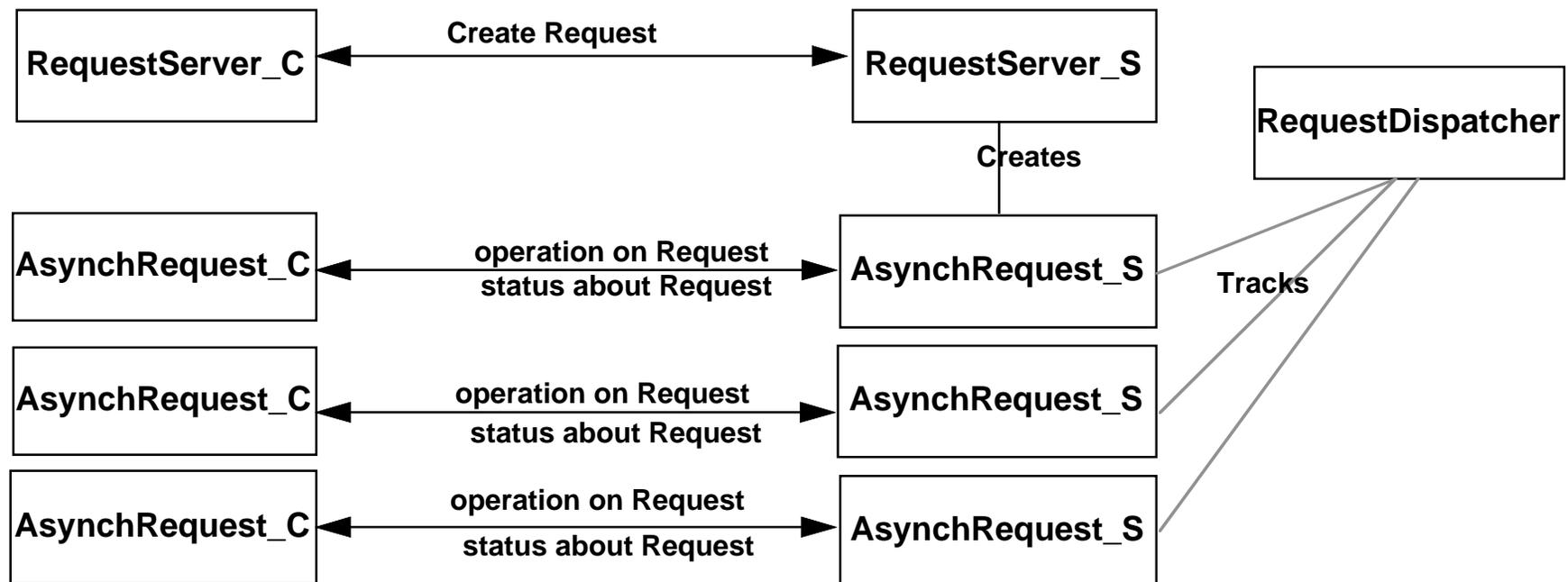
- communications, authorization, threads, etc.



# Software Design: logical view

Client

Server





# Object Model

The following object models will be reviewed:

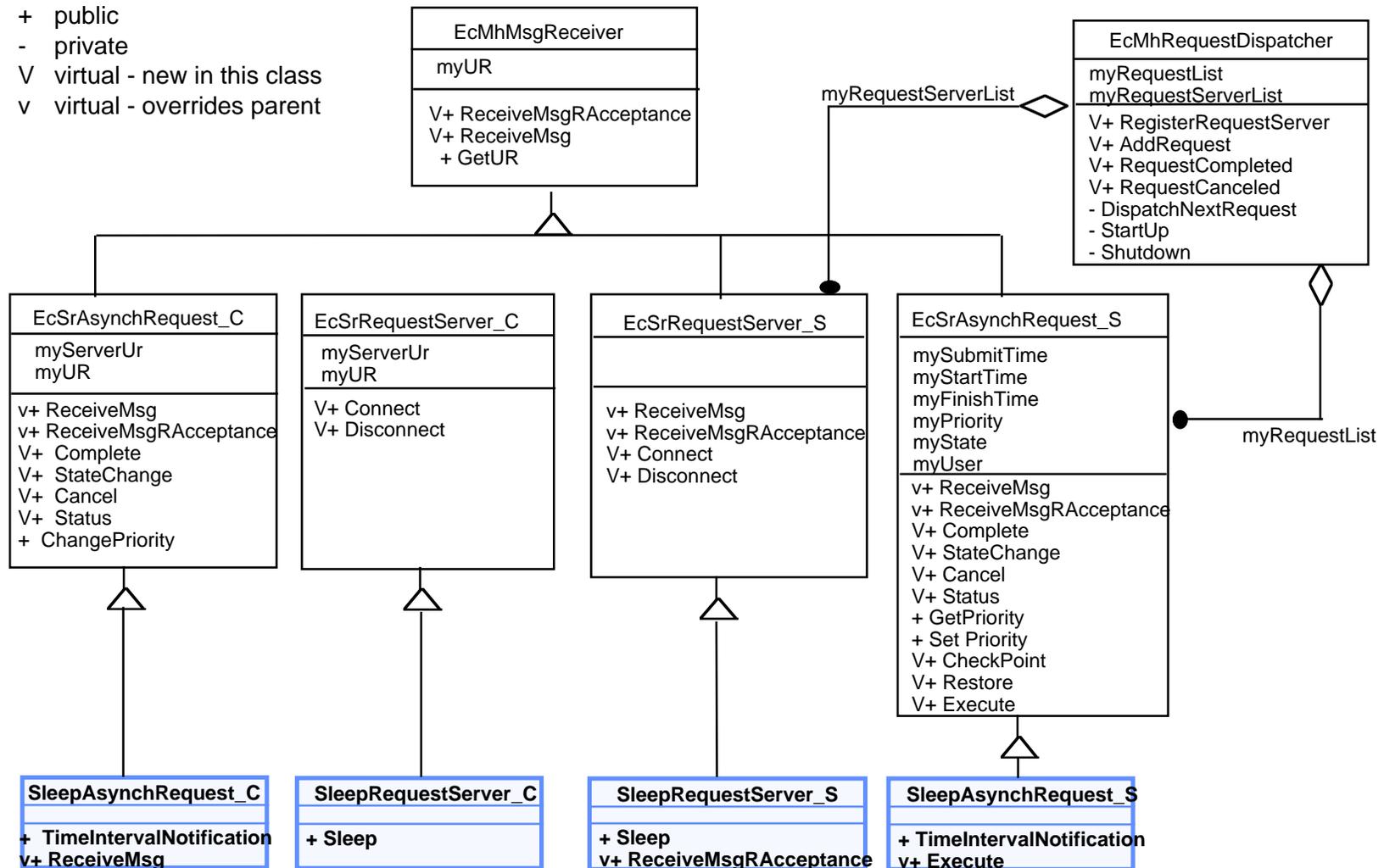
<u>Model Name</u>	<u>Document Reference</u>
ReIB_SRF	305-CD-028-002, Section 4.5.2
ReIB_SRFMessage	305-CD-028-002, Section 4.5.2

simplified object models included since they are easier to follow.



# Object Model: SRF

- + public
- private
- V virtual - new in this class
- v virtual - overrides parent



# Class Details



## EcSrRequestServer\_C

- **client-side server class - abstracts the connection to the server**
- **constructed with a server UR**
- **includes methods for Connected(), Disconnect(), Connect(), getServerUR()**
- **server developer specializes by adding new methods for operations**
  - **constructs message obj (EcUtStreamable) w/ values**
  - **calls SendMsgRAcceptance w/ server UR, msg, and response ptr**
  - **blocks until server response**
  - **response contains the EcSrAsynchRequest\_SUR to the EcSrRequestServer\_S**
  - **return server UR**
- **further specialized by client writer to handle client specific functionality**
  - **typically constructing EcSrAsynchRequest\_C of correct type**



# Class Details

## EcSrRequestServer\_S

- **server-side server class - abstracts the connection to the server**
- **constructed with server queue name & request dispatcher pointer**
- **includes implementations of Connect() & Disconnect()**
- **server developer must specialize ReceiveMsgRAcceptance()**
  - **dispatches EcUtStreamable based on type**
  - **if type not recognized, call parent method**
  - **return EcUtStreamable (response message)**
- **server developer adds new methods for server specific operations**
  - **create AsynchRequest\_S object and add to dispatch list**
  - **obtain UR from AsynchRequest\_S object and return**



# Class Details

## EcSrAsynchRequest\_C

- **client-side Request class - represents the currently active request**
- **constructed by server and sent to client**
- **constructor takes ErSrAsynchRequest\_S (learns request UR from this)**
- **includes implementations of Cancel(), ChangePriority(), Status()**
- **server developer must specialize ReceiveMsg() method**
  - dispatches EcUtStreamable based on type
  - if type not recognized, call parent method
- **server developer adds new methods for server specific operations**
- **need to provide implementations of StateChange() and Complete() callbacks**
  - typically done by client writer
- **client developer will specialize methods that handle callbacks for client specific functionality**
  - e.g, update display based on status of request



# Class Details

## EcSrAsynchRequest\_S

- **server-side Request class - represents the currently active request**
- **constructor should be specialized to take arguments relevant to request**
- **includes implementations of Complete(), and Status()**
- **server developer must write CheckPoint(), Restore(), and Execute()**
- **Execute()**
  - implements the actual request processing
  - called by SRF once when request is dispatched from queue
  - when done, call complete()
- **server developer adds new methods for server specific operations**
- **server developer must specialize ReceiveMsg() method if new methods added**



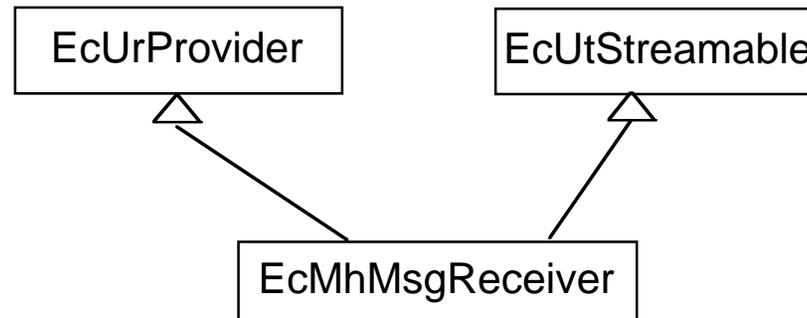
# Class Details

## EcSrRequestDispatcher

- **server-only object that handles the list of all requests**
- **dispatches requests (in a new thread) based on priority**
  - **calls object->Execute() to execute a request**
- **requests are managed via**
  - **AddRequest(EcSrAsynchRequest\*)**
  - **RemoveRequest(EcSrAsynchRequest\*)**
  - **RequestCompleted(EcSrAsynchRequest\*)**
  - **RequestCancelled(EcSrAsynchRequest\*)**
- **includes implementations of**
  - **ShutDown(), CheckPoint(), Restore()**



# Object Model: SRF Message



## Key Feature

- all MsgReceiver's are
  - UR providers
    - identified by URs for sending/receiving messages
  - streamable objects
    - can be sent as messages or stored persistently



# Class Details

## EcUtStreamable

- class which can flatten/unflatten itself to/from a stream
- automatically generated from \*.inp files

```
MESSAGE MsgIntervalNotif
CLASS_ID TestMsgIntervalNotification IN EcSrClassID.h
ATTRIB EcTInt TimeInterval
```

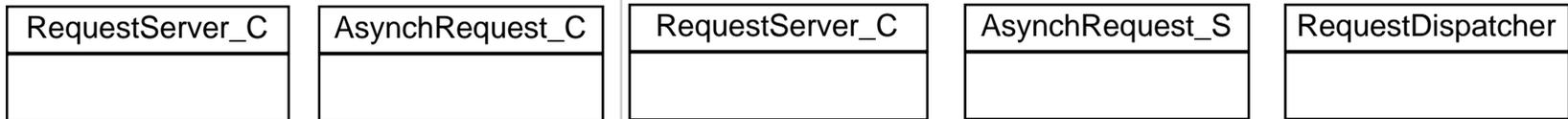


# SRF Specialization

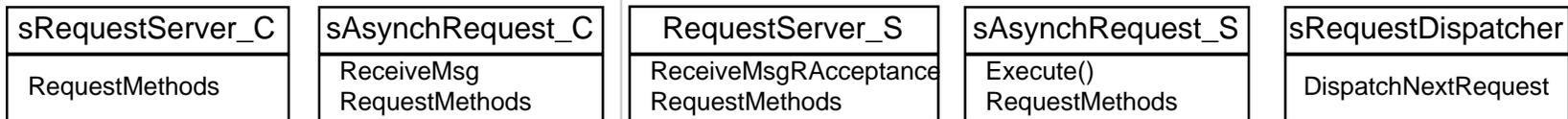
## Client Side

## Server Side

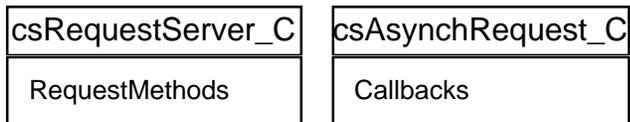
**SRF**



**Server  
Writer**



**Client  
Writer**





# Dynamic Model

Two simplified event traces follow. They show the client & server viewpoints of SRF.

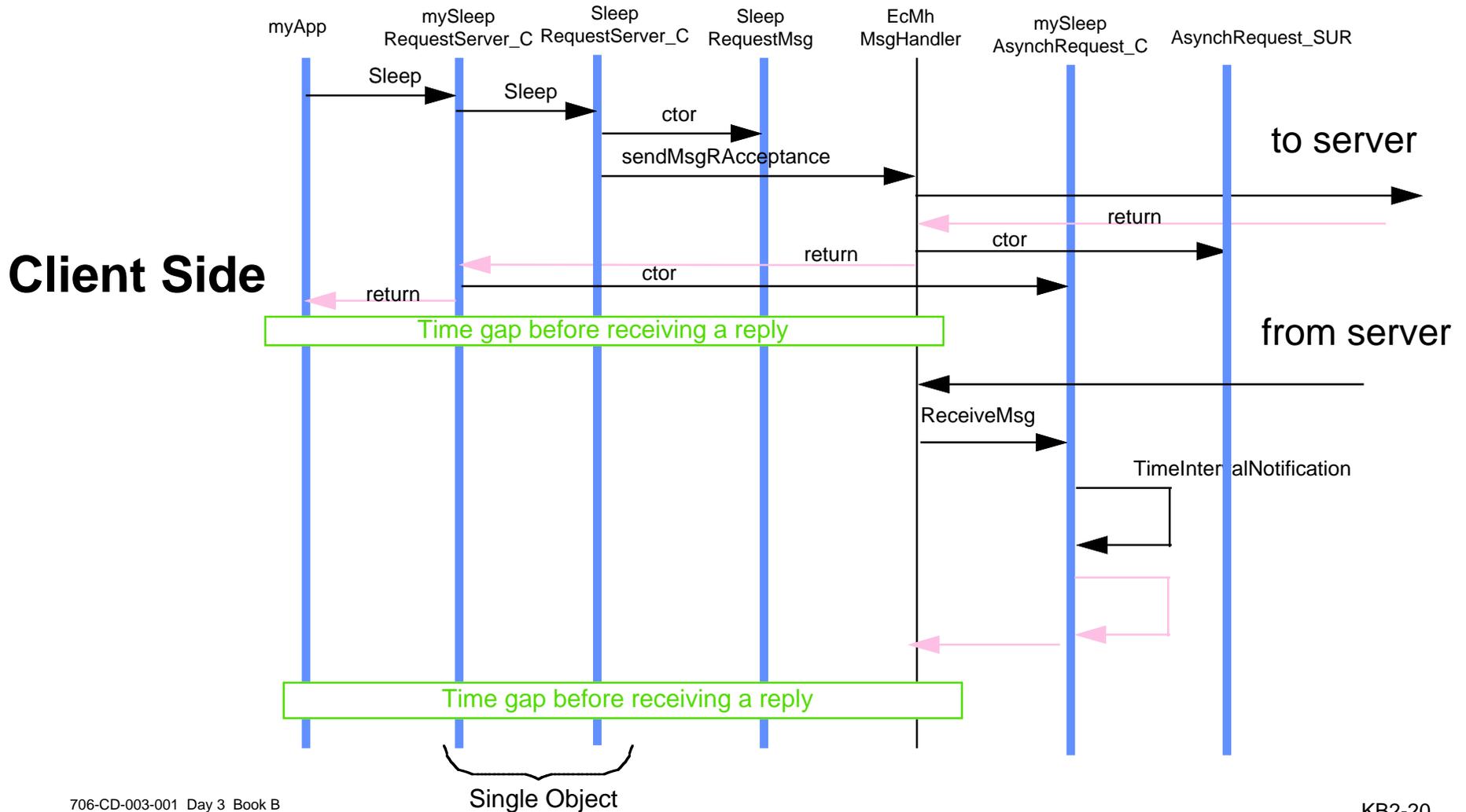
- blue vertical bars represent objects created by application developers
- black vertical bars represent objects that are part of SRF
- black lines represent method calls or function calls
- pink lines represent returns (e.g, returning control to the caller)

These event traces are for a hypothetical server that provides a “sleep” service.

- the client makes a sleep() request
- the server notifies the client at each notification interval



# SRF Scenario





# Client Trace Highlights

## Preconditions

- SRF was initialized
- client constructed `mySleepRequestServer_C` already and is connected to the server

## Steps

- client application calls `sleep()` method
- `mySleepRequestServer_C::sleep()` calls parent `sleep()` method
- `SleepRequestServer_C::sleep()`
  - constructs sleep message
  - sends with acceptance (will wait for server response)
  - the SRF Mh layer sends the message to the server
  - later, server replies (the message is a `AsynchRequest_SUR`)
  - Mh layer constructs `AsynchRequest_SUR` object
- `mySleepRequestServer_C` constructs `mySleepAsynchRequest_C` from `SUR`
- client application gets pointer to `mySleepAsynchRequest_C` object



# Client Trace Highlights

## When the timer expires on the server

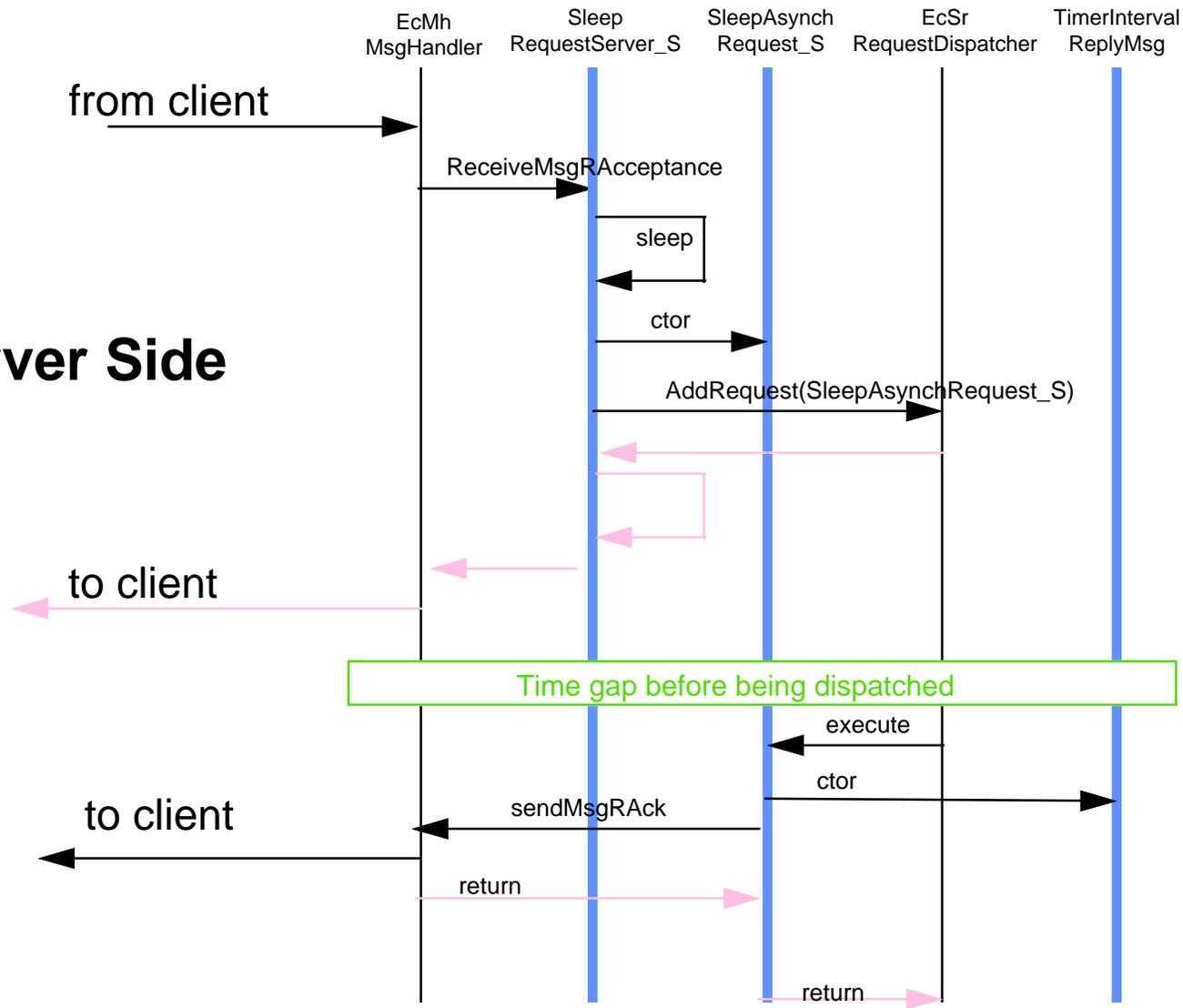
- client receives IntervalNotif message
- SRF Mh layer calls ReceiveMsg on mySleepAsynchRequest\_C object
- SleepAsynchRequest\_C::ReceiveMsg
  - dispatches to correct method based on message type
  - client app writer could specialize SleepAsynchRequest\_C to do something application specific
- repeat until done

Eventually, complete() method will be called on client object



# SRF Scenario

## Server Side





# Server Trace Highlights

## Preconditions

- SRF was initialized
- server constructed `SleepRequestServer_S`
- client has already connected to server

## Steps

- server receives sleep message from client
- SRF Mh layer dispatches to `SleepRequestServer_S` object
- `SleepRequestServer_S::ReceiveMsgRAcceptance`
  - dispatches to `sleep()` method based on message type
  - construct `SleepAsynchRequest_S` & adds to Request queue
  - returns `AsynchRequest_SUR` (as `EcUtStreamable`) to SRF Mh layer
  - message is sent to client



# Server Trace Highlights

## When SRF dispatches the SleepAsynchRequest

- **RequestDispatcher calls SleepAsynchRequest->Execute()**
- **Execute()**
  - **constructs a InvervalNotif message**
  - **sends it via sendMsgRAck (waits for reply)**
  - **repeats previous step multiple times**
  - **eventually completes (returns control to RequestDispatcher)**



# Client Trace Pseudo-Code

```
//called by client application
mySleepRequestServer_C::Sleep(...)
{
    AsynchRequest_SUR *SUR;
    SleepRequestServer::Sleep(..., UR)
    a = new mySleepAsynchRequest_C(UR)
    return a;
}
```

```
SleepRequestServer_C::Sleep(int time, int notifInt, SUR*& UR)
{
    sleepMsg m(time, notifInt);
    //Use global SRF message handler to send
    theHandler->SendMsgRAcceptance(GetServerUR(), //UR of RequestServer_S
                                   myUR,
                                   m,
                                   UR);
}
```



# Client Trace Pseudo-Code (2)

```
//SRF internals
//called as theHandler by clients
MsgHandler::SendMsgRAcceptance(MsgReceiverUR& to,
                               MsgReceiverUR& replyTo,
                               UtStreamable& msg,
                               UtStreamable*& acceptMsg)
{
    //create a pending message object that will handle the message &
    // it's response.
    PendingMsg m;

    m.SendMsgRAcceptance(to, replyTo, msg, acceptMsg);
}
```



# Client Trace Pseudo-Code (3)

```
//Send message, wait for response
PendingMsg::SendMsgRAcceptance(MsgReceiverUR& to,
                                MsgReceiverUR& replyTo,
                                UtStreamable& msg,
                                UtStreamable*& acceptMsg)
{
    MsgEnvelope env(to,
                    ProvideUR(), //my UR, used for the callback
                    replyTo,
                    msg)

    ...flatten envelope....
    ...send using EcMp* classes...
    //now wait for callback. the response message will go to this
    //object (via ReceiveMsg). The ack will be handled by HandleAck() method
    myCond.Wait()
    acceptMsg = myStreamableStore;
}
```



# Client Trace Pseudo-Code (4)

```
//Called by message passing layer on receipt of message
PendingMsg::ReceiveMsg(UtStreamable& newMessage, ClientInfo& info)
{
    myStreamableStore = newMessage;
}

//Called by message passing layer to ack messages
PendingMsg::HandleAck(..., EctBoolean failedFlag)
{
    if (failedFlag) {
        myMsgState.SetFailed();
    }

    myCond.Signal() //wake up the waiting method
}
```



# Client Trace Pseudo-Code (5)

```
//called when callback occurs
SleepAsynchServer_C::ReceiveMsg(UtStreamable& msg,
                                ClientInfo& info)
{
    if (msg.GetClassID() == IntervalNotifID) {
        //Call appropriate method. Since virtual, could have been
        //overridden by client writer
        TimeIntervalNotification(msg.GetTimeInterval());
        return;
    }
    //unknown, let SRF handle
    EcSrAsynchRequest_C::ReceiveMsg(msg, info);
}

//client writer wrote this method
myAsynchRequest_C::TimeIntervalNotification()
{
    ...do something client specific...
}
```



# SRF/PF Issues

## SRF clients are really servers (in the DCE sense)

- **PF unmanaged server is too heavy**
  - includes FTP, principal name, keytab, CDS name, etc
  - need to create lightweight PF server for SRF
- **SRF Security**
  - clients normally inherit identity from user
  - servers normally use keytab (as in PF unmanaged server)
  - this is a problem for an SRF client
    - needs to use user's identity, not identity from keytab file
- **these (minor) issues are currently being addressed w/ Release A**