Out-of-Process Components

Jim Fawcett
CSE775 - Distributed Objects
Spring 2011

Out-of-Process Architectural Diagram

- An out-of-proc component has essentially that same structure as an in-proc component, except that:
  - Outproc is built as a console or windows application, not a dll.
  - The component has command line processing that will either:
    - Register the component
    - Unregister the component
    - Run the component server
  - The main or winmain:
    - creates the class factory
    - calls CoRegisterClassObject to put an entry into the running object table (ROT) telling COM that it is running and stores the factory pointer.
  - When a client calls CoCreateInstance COM gets the class factory pointer from the ROT and creates the component instance.

local.cpp

struct IUnknown

virtual Hresult QueryInterface(...) = 0;
virtual ULONG AddRef( ) = 0;
virtual ULONG Release( ) = 0;

struct ISum

virtual void Fx( ) = 0;

struct IClassFactory

virtual HRESULT CreateInstance(...) = 0;
virtual HRESULT LockServer(...) = 0;

Class CInsideDCOM

Attribute:
long m_cRef;

Operation:
CA( )
~CA( );
virtual HRESULT QueryInterface(...)
virtual ULONG AddRef( );
virtual ULONG Release( );
virtual void Sum(int,int,int*);

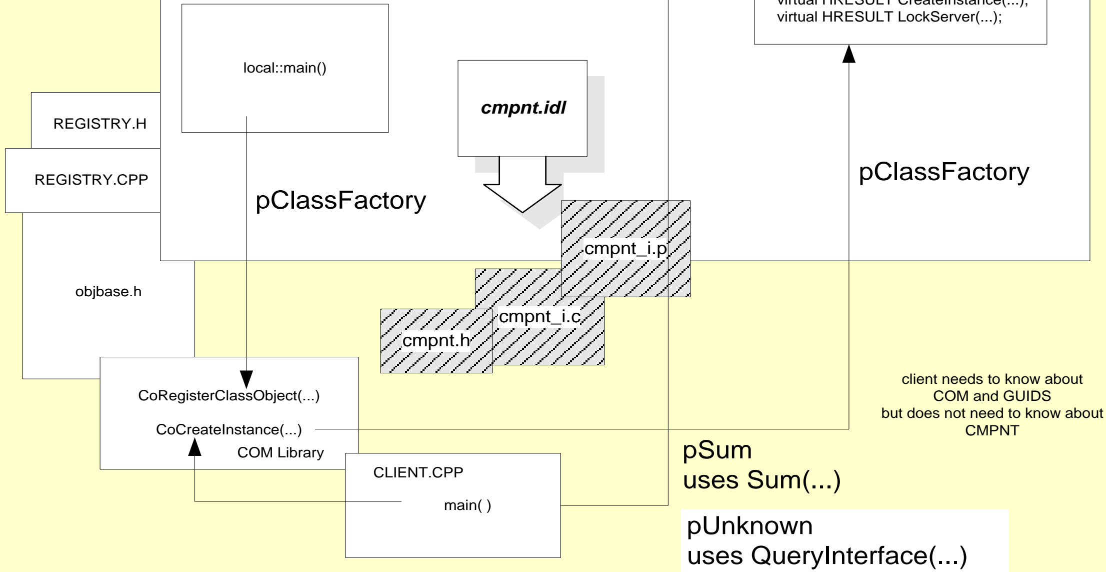
Class CFactory

Attribute:
long m_cRef;

Operation:
CFactory( )
~CFactory( );
virtual HRESULT QueryInterface(...);
virtual ULONG AddRef( );
virtual ULONG Release( );
virtual HRESULT CreateInstance(...);
virtual HRESULT LockServer(...);

OUTPROC Component

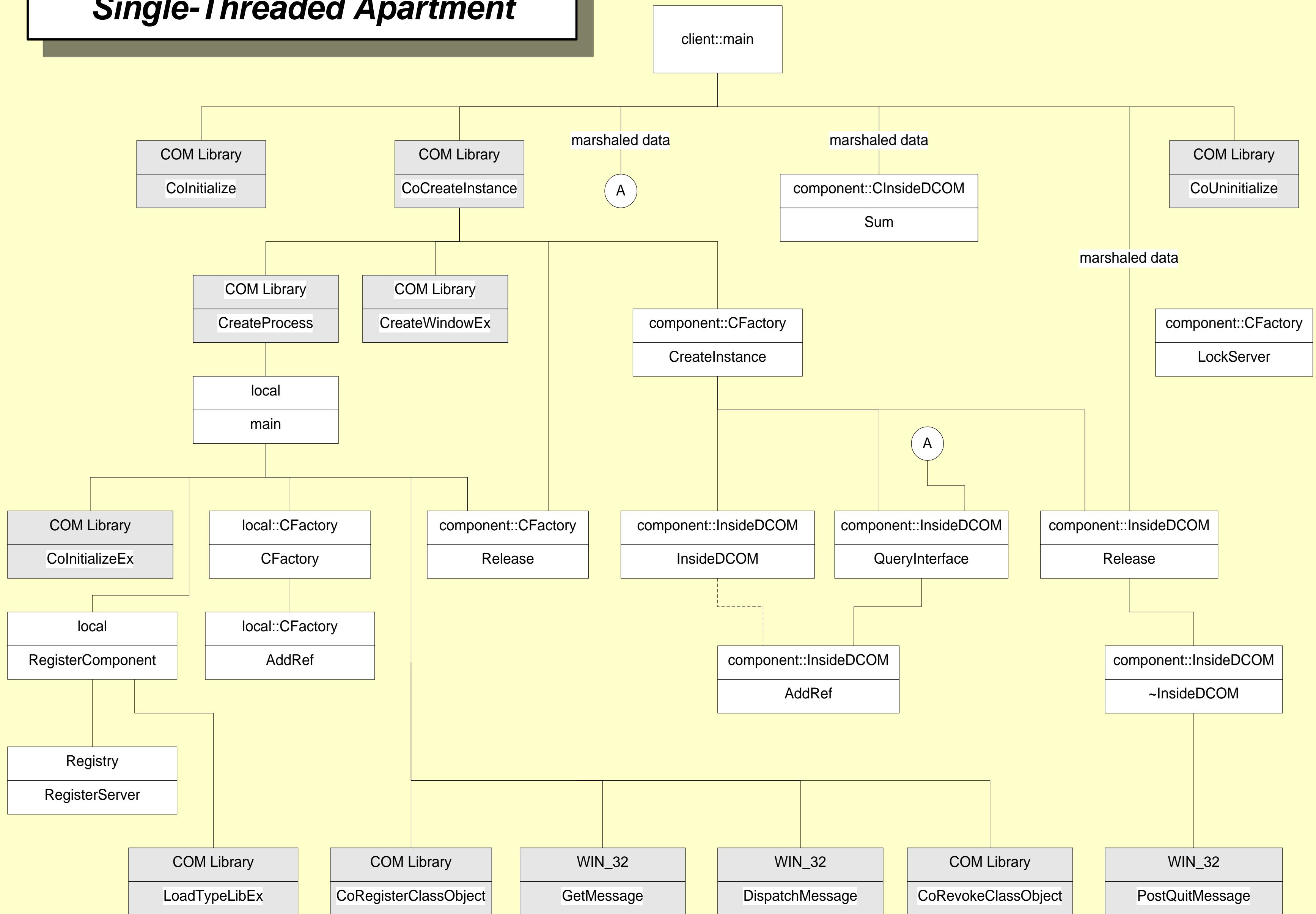
- Server creates a factory
- Passes factory pointer to COM in call to CoRegisterClassObject
- COM creates component instance when client calls CoCreateInstance
- Client can then call instance



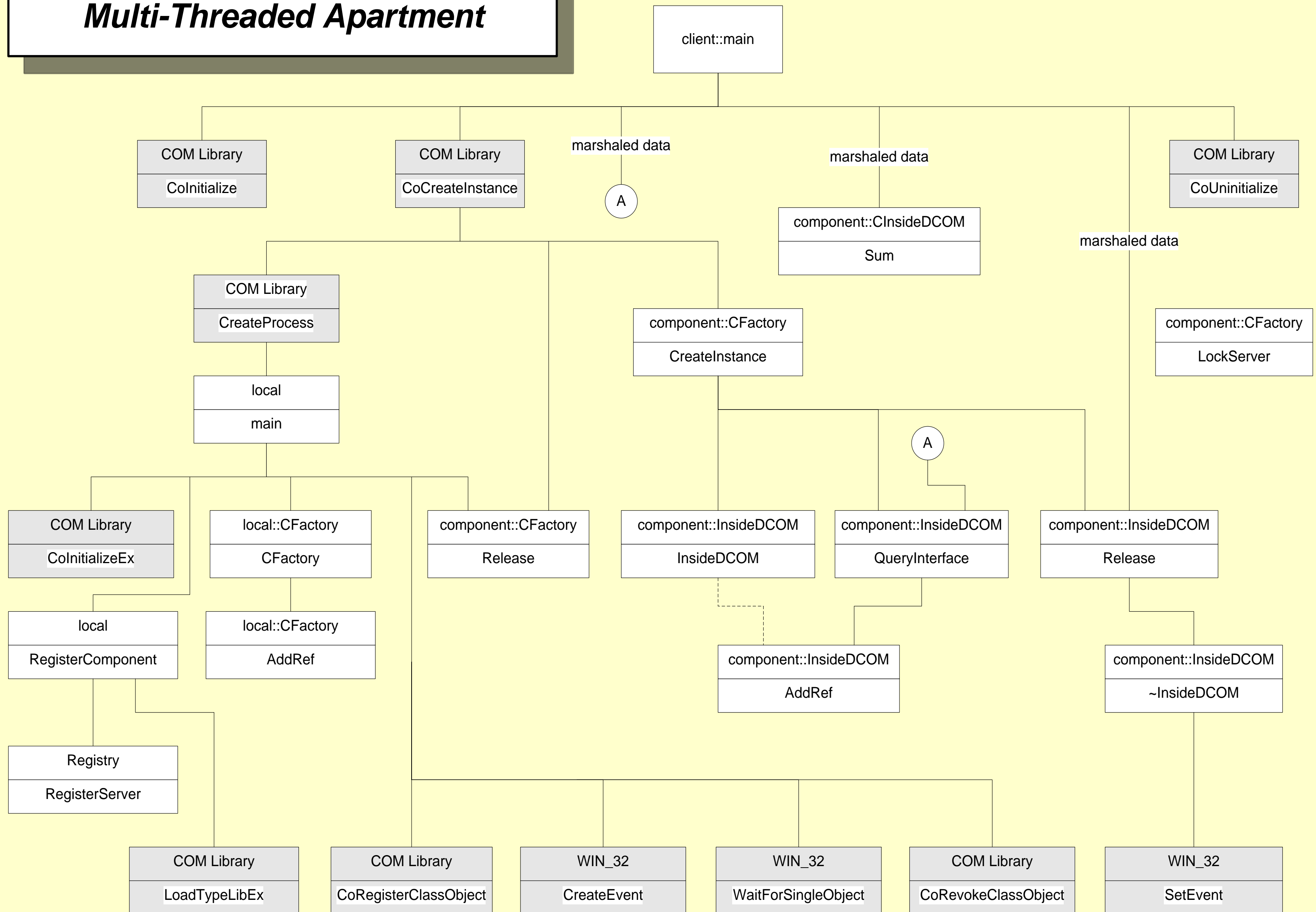
Structure Chart

- This diagram doesn't show the register and unregister processing for simplicity.
- The server is registered by the command:
  component /RegServer
- Once the component is registered, when the client calls CoCreateInstance COM will start the server process if it is not already running. That causes main to start the class factory and store a pointer to it in the Running Object Table (ROT).
- COM uses the pointer to create an instance and returns a pointer to the client.
- The client uses the component through its interface, then calls release so COM can shut down the server.

OutprocST - Structure Chart
Single-Threaded Apartment



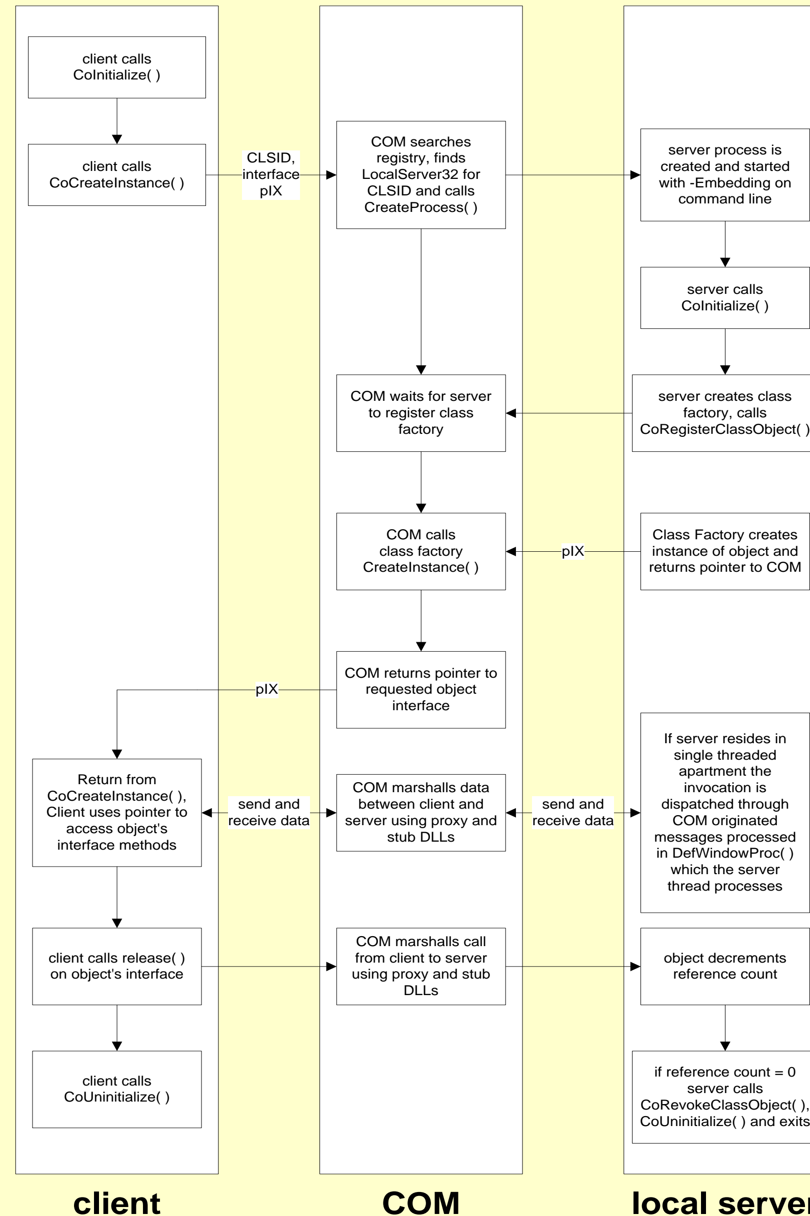
OutprocMT - Structure Chart
Multi-Threaded Apartment



Activation Diagram

- The diagram on the next page is an elaborated event trace diagram.
- It shows the separate, cooperating actions of the client, COM, and the Component to:
  - Create the component instance
  - Use it
  - Shut it down
- Note that one new element has been added. Communication between the client and component must be marshalled, e.g., using interprocess communication via remote procedure calls (RPCs).
- We will discuss the marshalling process in the next section.

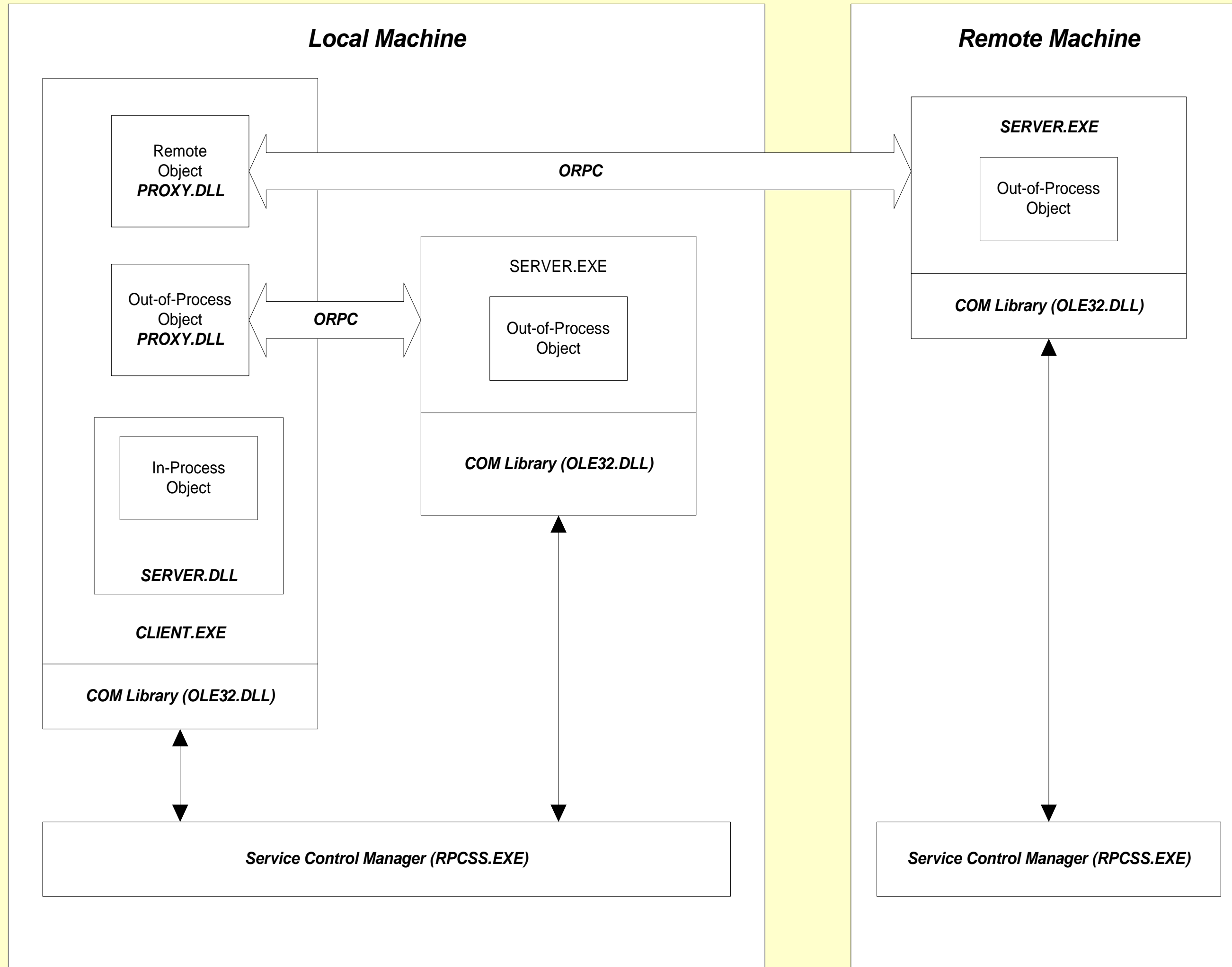
Activation diagram



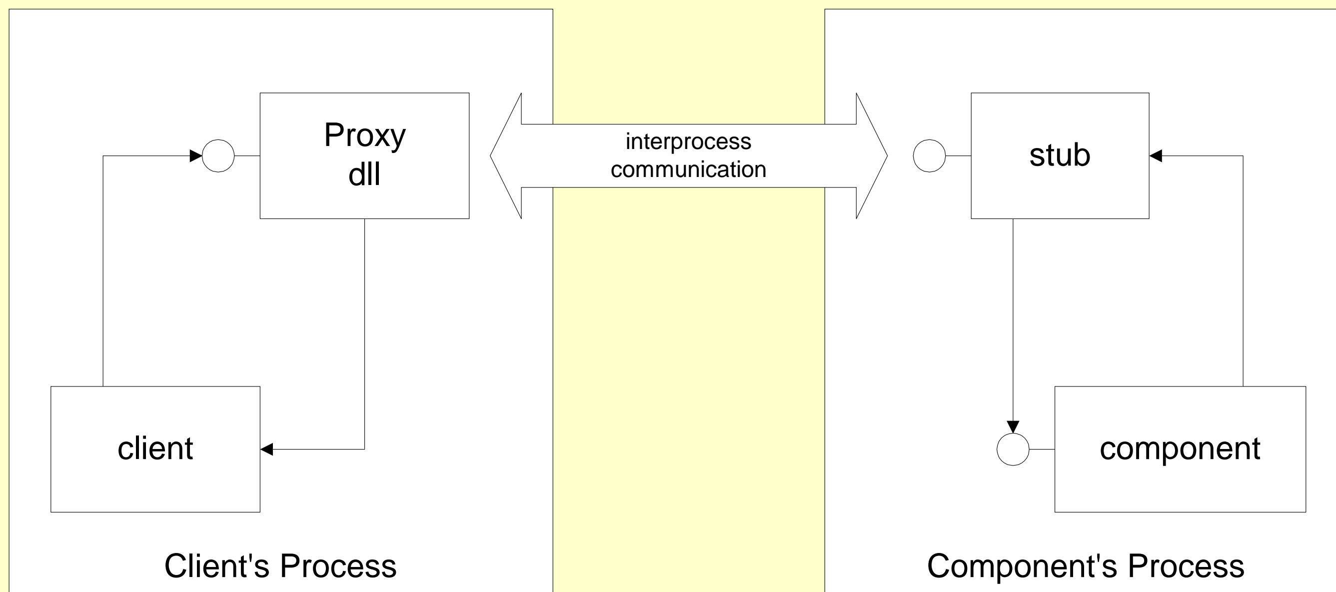
Marshaling

- All communication between client and an out-of-process component is accomplished through marshaling.
- Marshaling is the activity of decomposing data into a byte stream which is sent over an inter-process communication mechanism, then recomposing in the receiving process.
- COM's method of interprocess communication is the Remote Procedure Call (RPC). The client makes a synchronous call to a proxy in the client process. The proxy marshals the data and sends it to a stub in the receiving process where it is processed according to the semantics of the (remote) procedure. Any results are marshaled back to the client's proxy which then creates the fiction of a procedure return.
- It is the job of the Service Control Manager (SCM) to find the server and instantiate an object and connect the proxy and stub.

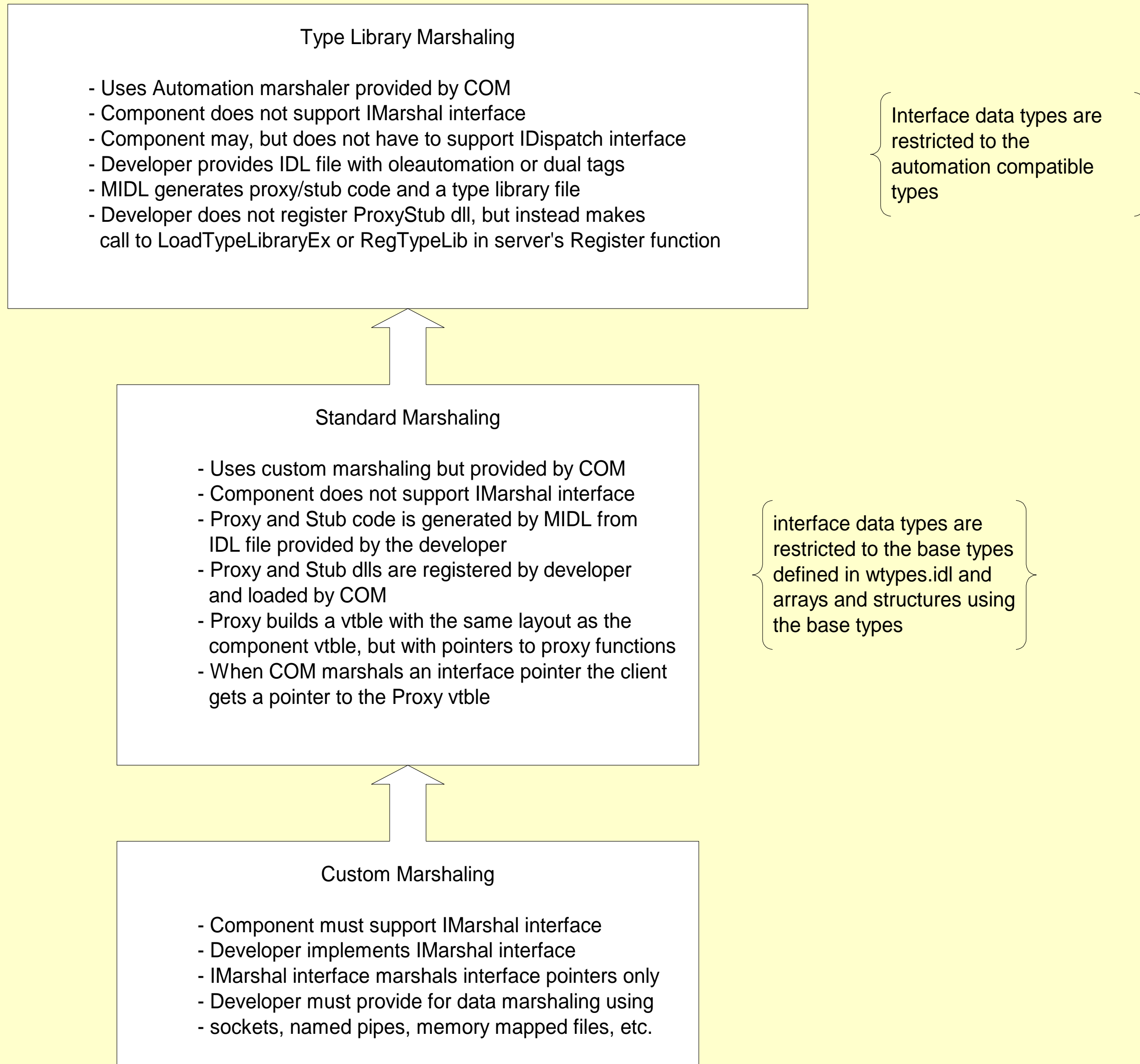
Component Activation



Marshaling Architecture



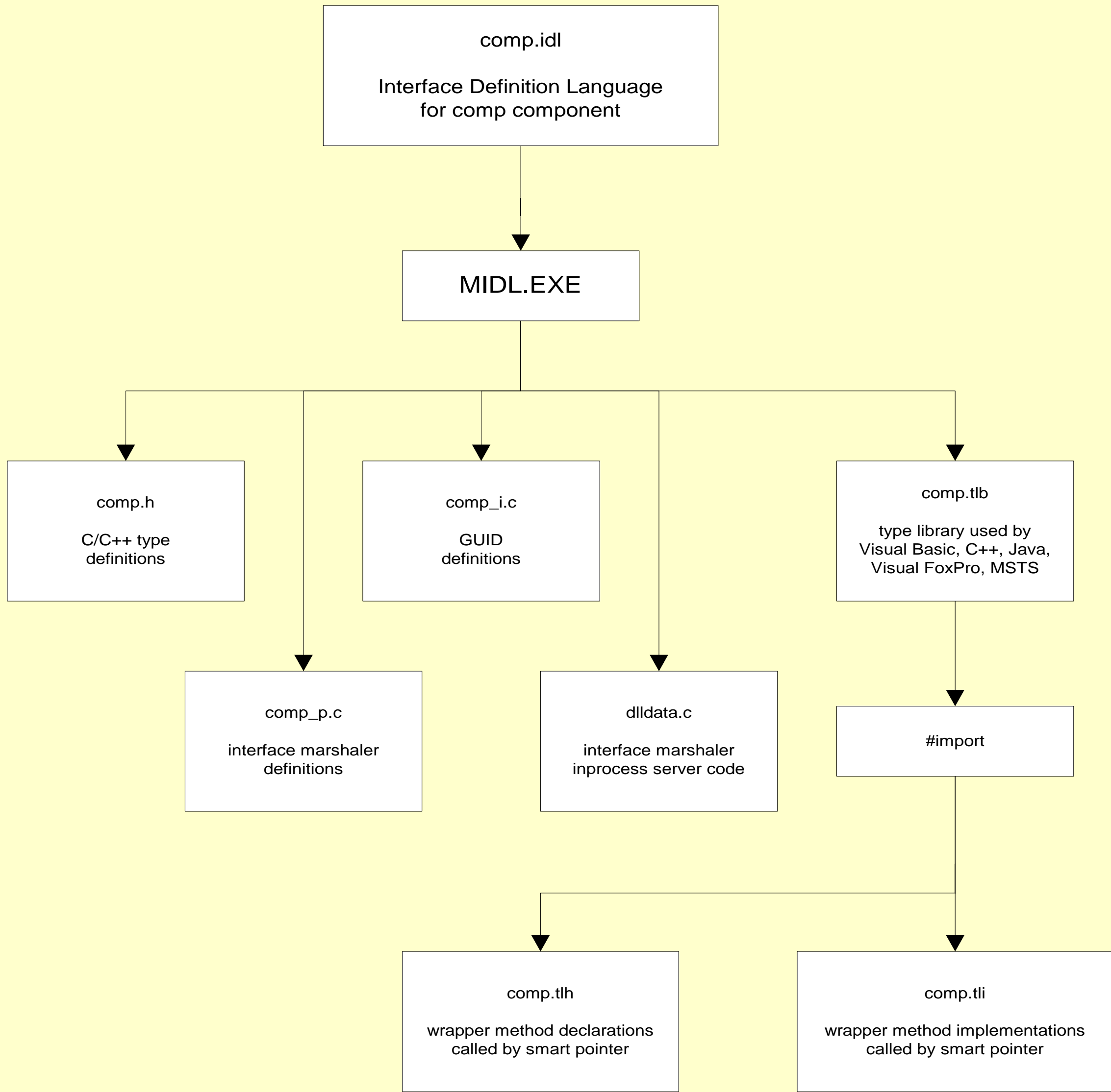
Marshaling Layers



MIDL Compiler

- Most COM components describe their interfaces in IDL rather than write an IFACE type header for interface declarations and an IFACE implementation to define GUIDs.
- IDL is compiled by the Microsoft Interface Definition Language (MIDL) compiler to generate the components shown in the diagram on the next page.
- If you include the IDL file in your project, you can right click on the IDL file (in file view) and select the compile item to process your interface descriptions.

MIDL chart



Threading Models and Apartments

- COM supports the interaction between client and server even if they make different assumptions about threading.
  - A component that does not want to handle multiple accesses by concurrent threads specifies that it wishes to operate in a Singe Threaded Apartment (STA). Any single process can support one or more STAs.
  - A component that will handle multiple accesses by concurrent threads, by synchronizing access to its global and static data, specifies that it wishes to operate in a Multiple Threaded Apartment (MTA). A process can support only one MTA.
  - An out-of-process component, or client for an in-process component makes this specification by the arguments it passes to CoInitializeEx.

Creating Apartments

- A single threaded apartment is created on a call to

```
CoInitialize(NULL)
```

or

```
CoInitializeEx(NULL,COINIT_APARTMENT_THREADED)
```

- A multi-threaded apartment is created on a call to

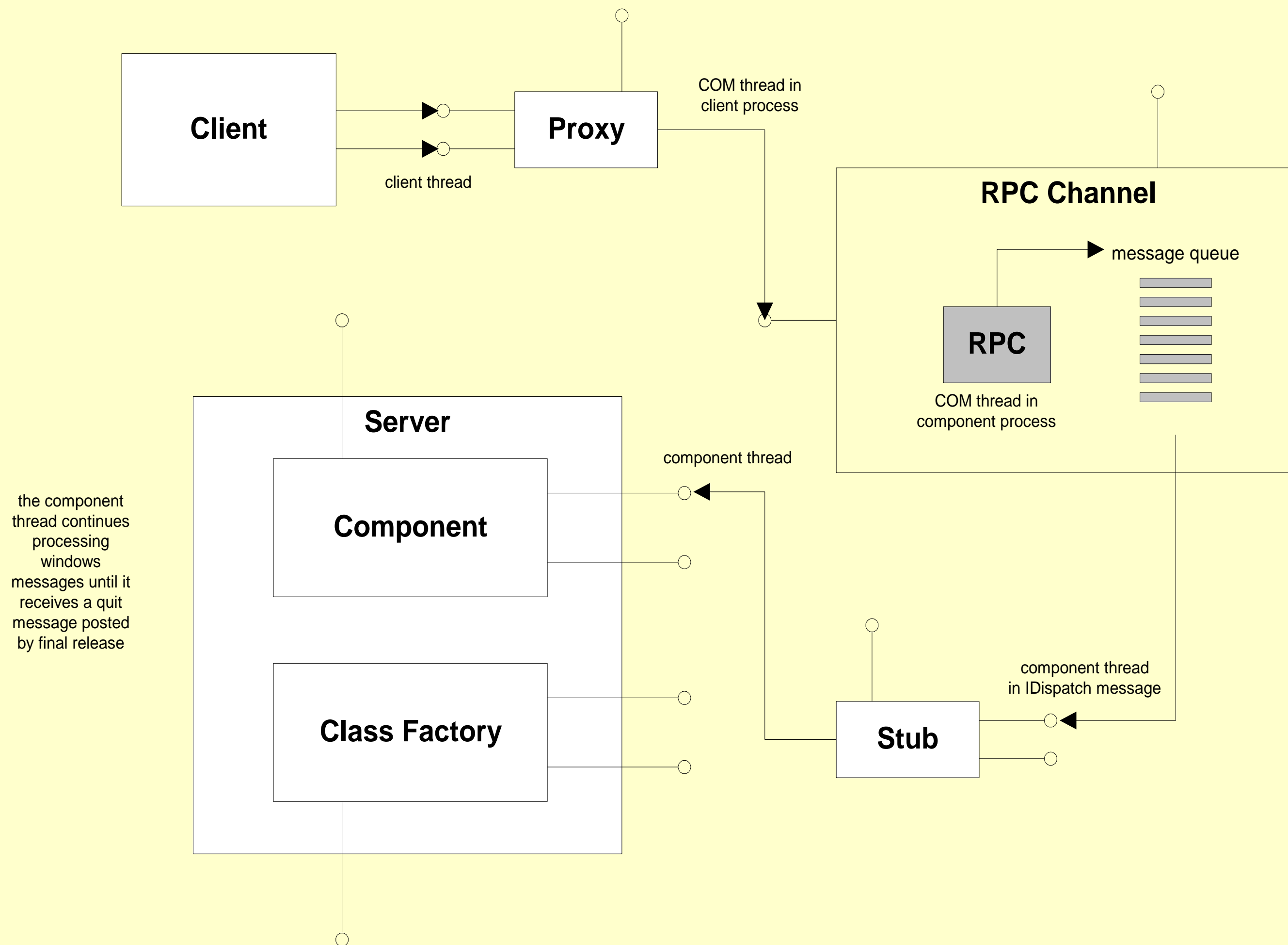
```
CoInitializeEx(NULL,COINIT_MULTITHREADED)
```

- Since an Out-of-Proc component runs in its own process, its server must call CoInitialize(Ex).

STA Remoting Architecture

- When a client calls into a server residing in an STA the call is serviced by the server thread.
- A local or remote call is made, on the client's behalf, by an RPC thread created by the server's stub. The RPC thread deposits a request for service from the desired interface function on a windows message queue, created when the server was started.
- The server's thread – the only thread allowed in its STA – retrieves the message, executes the function, and packages the return data for transmission back to the client.
- When multiple clients make concurrent calls they are simply enqueued for service by the single STA thread. Thus the server never has to handle multiple concurrent threads.

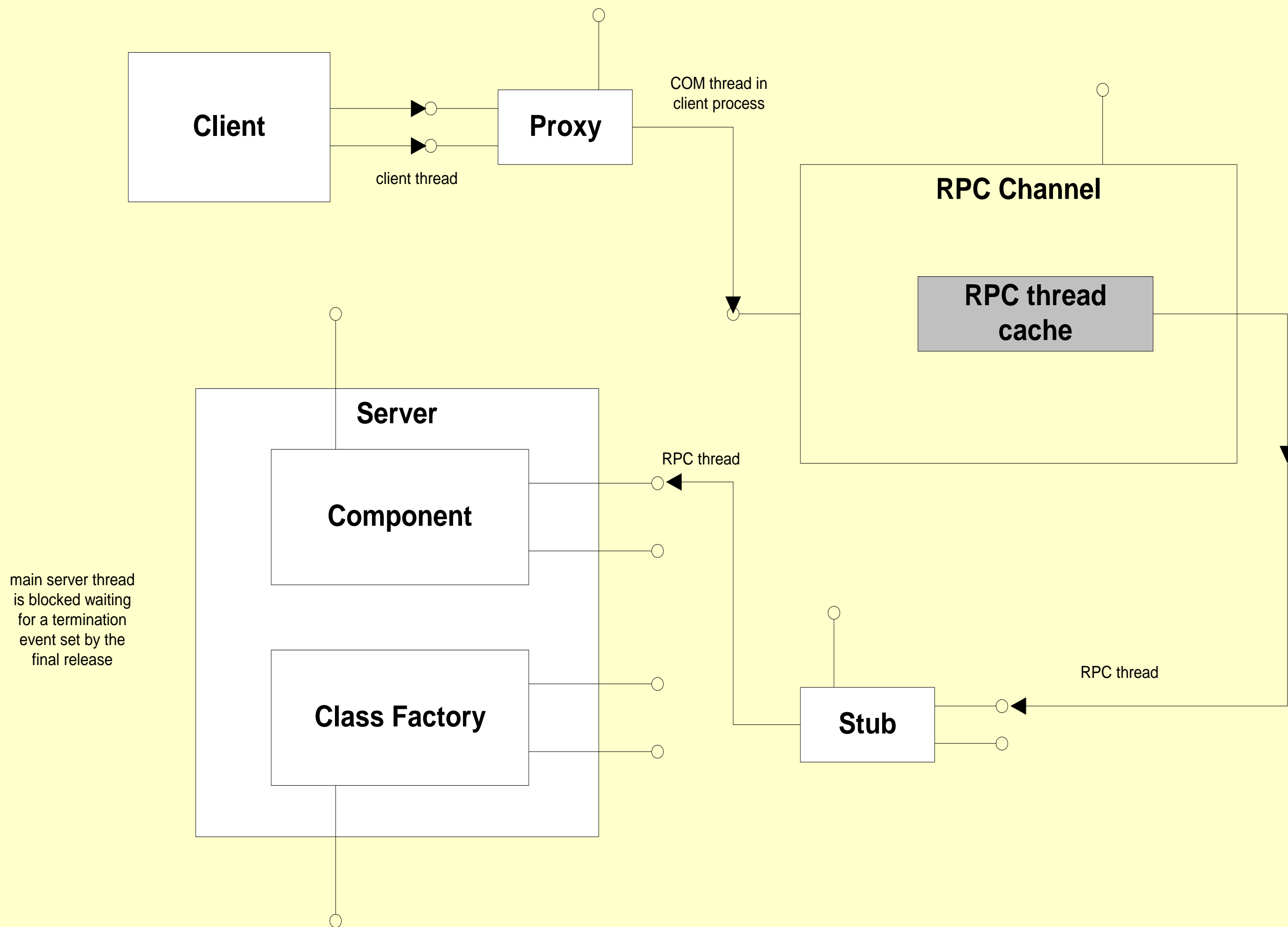
STA Remoting Architecture



MTA Remoting Architecture

- When a server resides in an MTA it must be prepared to deal with concurrent access by several threads. It does this by synchronizing access to all global and static data.
- The RPC channel creates a pool of RPC threads for processing client requests.
- When a client request arrives, an RPC thread is dispatched to process the requested interface function, even if another thread is already active in the MTA.

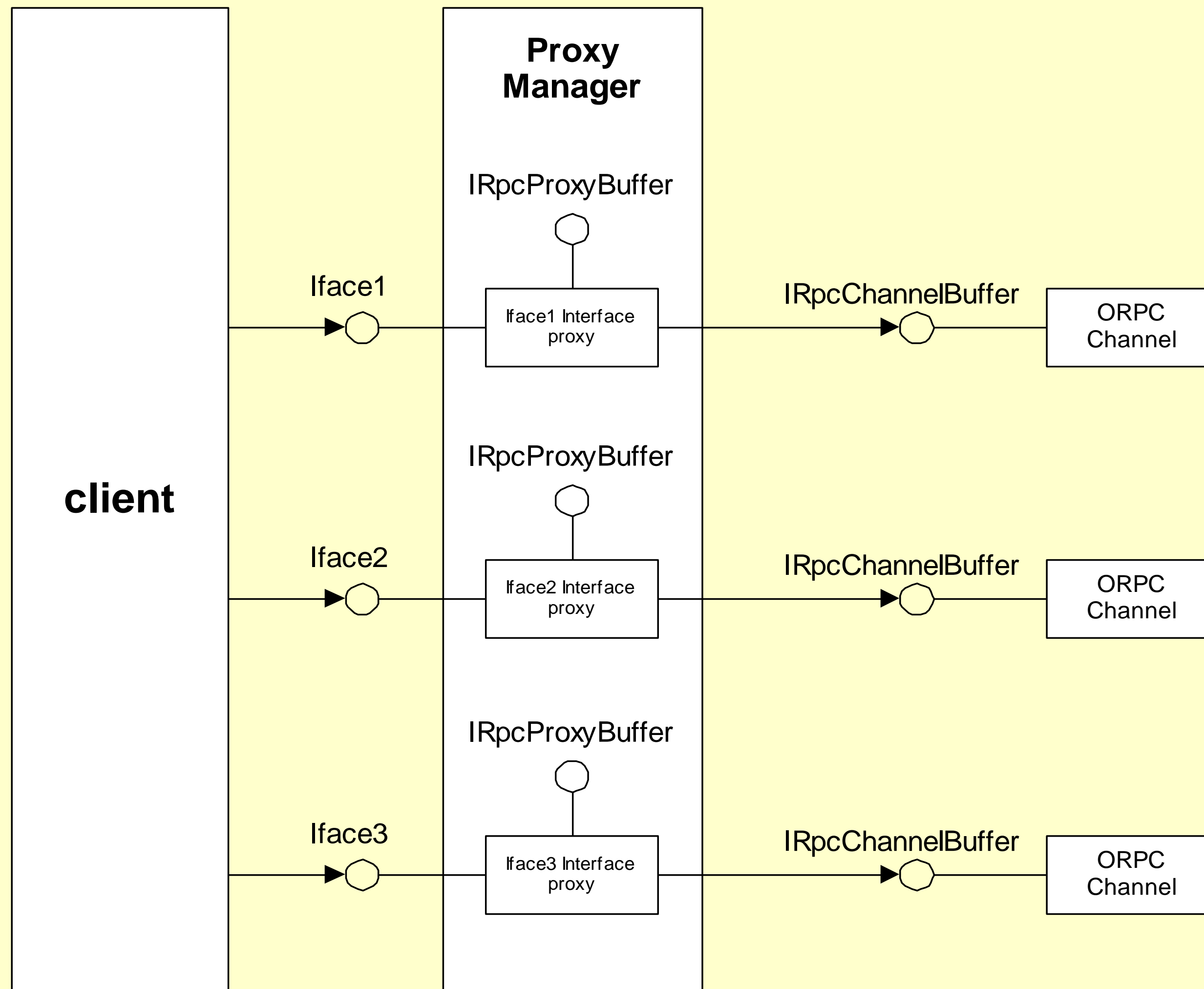
MTA Remoting Architecture



Remoting Architecture - Proxies

- The next two diagrams show how the RPC channel manages proxies and stubs for client server interaction.
- On the client side the RPC channel creates a proxy manager that creates, on demand, a proxy for each interface the client accesses.
- Each proxy has an RPC buffer to use for enqueuing bytes to be sent to the client, as part of the marshaling process.
- The proxy manager uses COM aggregation to hand the proxy interface to the client. Thus, the client views all of the interfaces as belonging to a single entity, and QueryInterface works as expected.

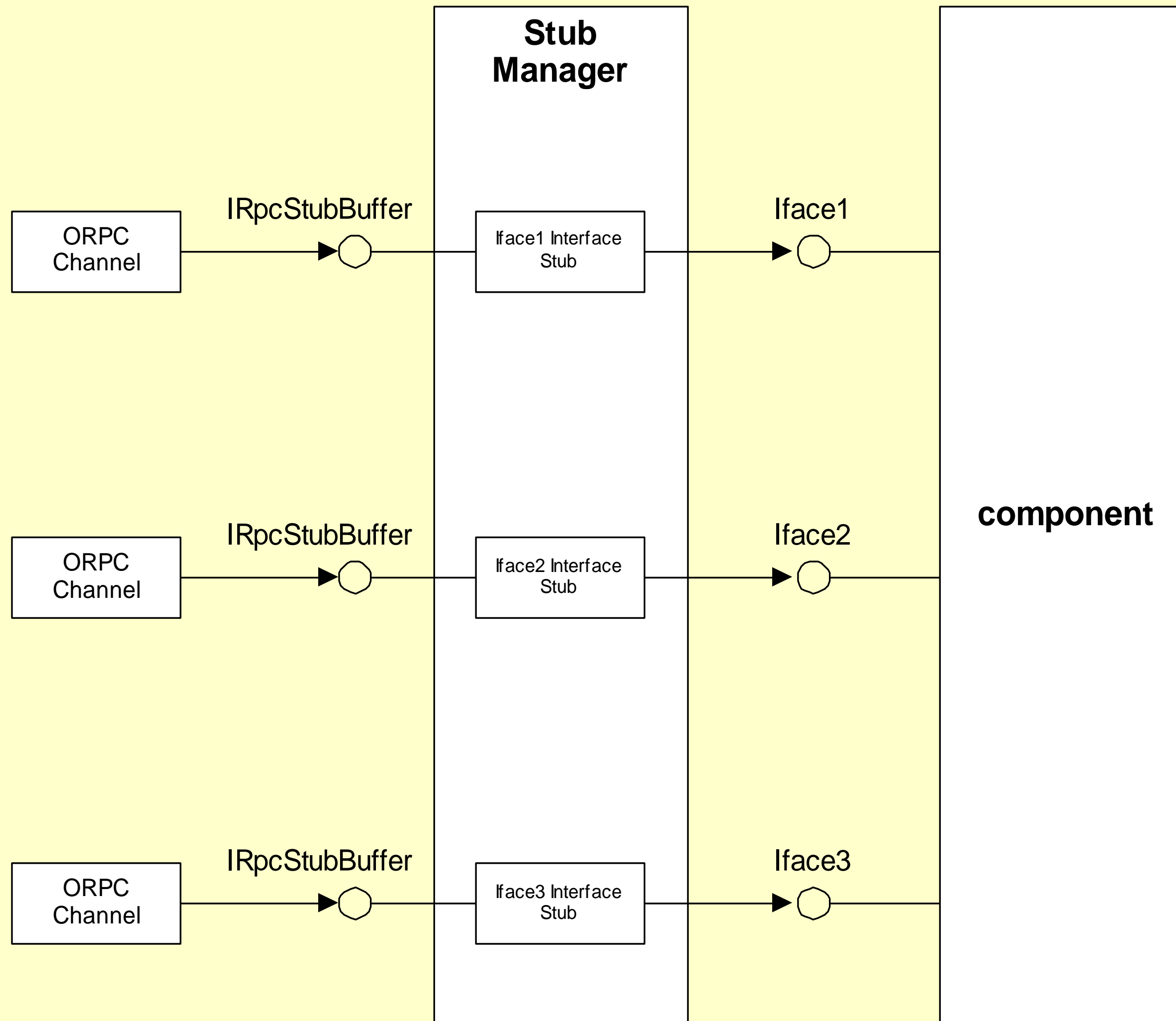
Proxy Manager



Remoting Architecture - Stubs

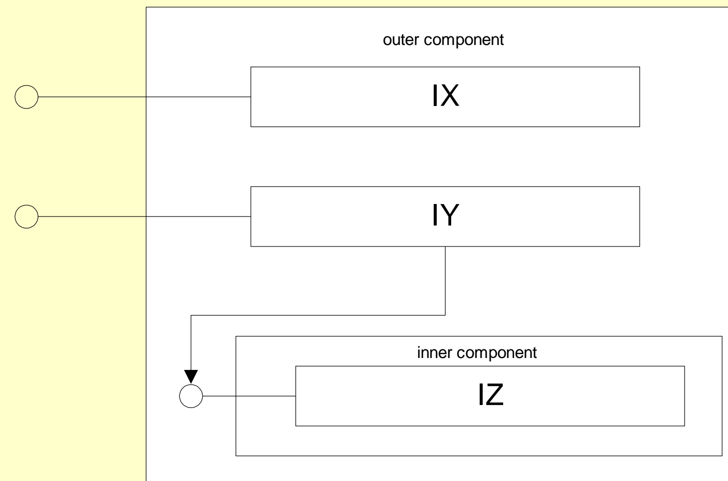
- On the server side a stub manager is created when the server is activated.
- The stub manager creates a stub, on demand, for each interface supported by the server.
- The server does not need to view the stub manager as a single entity, since it will not be calling QueryInterface on the client. So, COM aggregation of the stubs by the stub manager is not required.

Stub manager



COM's Reuse Mechanisms: Containment

- COM defines component containment which has semantics of C++ aggregation but can be composed at run time.
- With containment the reusing COM object loads an existing component and implements part of its own interface by delegating calls to the contained component.



Implementing Containment

- Containing component class:
  - provides an interface matching the contained classes interface and delegates calls to the inner interface (optional).
  - provides an init( ) function which calls CoCreateInstance(…) on the contained component.
  - Declares a pointer member to hold the pointer to inner interface returned by CoCreateInstance(…).
  - Outer component's class factory calls init( ) in CreateInstance(…) function.
- Client:
  - no special provisions.
- Inner Component:
  - no special provisions

COM's Reuse Mechanisms: Aggregation

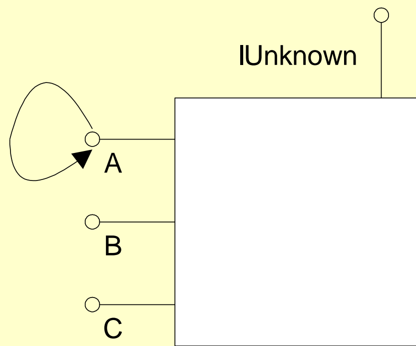
- What COM chose to define as aggregation is unfortunately quite different than C++ aggregation.

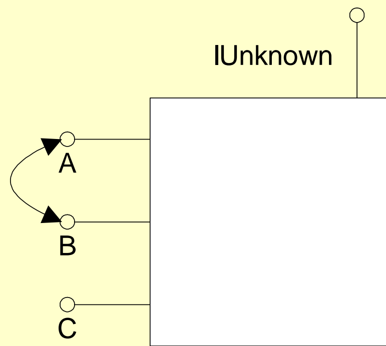
- With COM aggregation the aggregating class forwards interface of the reused existing class to the client by delivering a pointer to the aggregated interface.
  - This complicates implementation of the inner IUnknown since the usual COM policy for interfaces must still be carried out.
  - The result is that, in order to be aggregate-able a component must implement two IUnknown interfaces

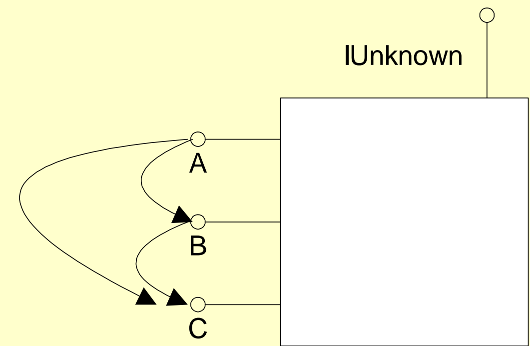
COM Interface Policy

- COM requires that:
  - all calls to QueryInterface for a given interface must return the same pointer value
  - the set of interfaces accessible from QueryInterface must be fixed
  - if a client queries for an interface through a pointer to that interface the call must succeed
  - if a client using a pointer for one interface successfully queries for a second interface the client must be able to successfully query through the second interface pointer for the first interface
  - if a client successfully queries for a second interface and, using that interface pointer successfully queries for a third interface, then a query using the first interface pointer for the third interface must also succeed.

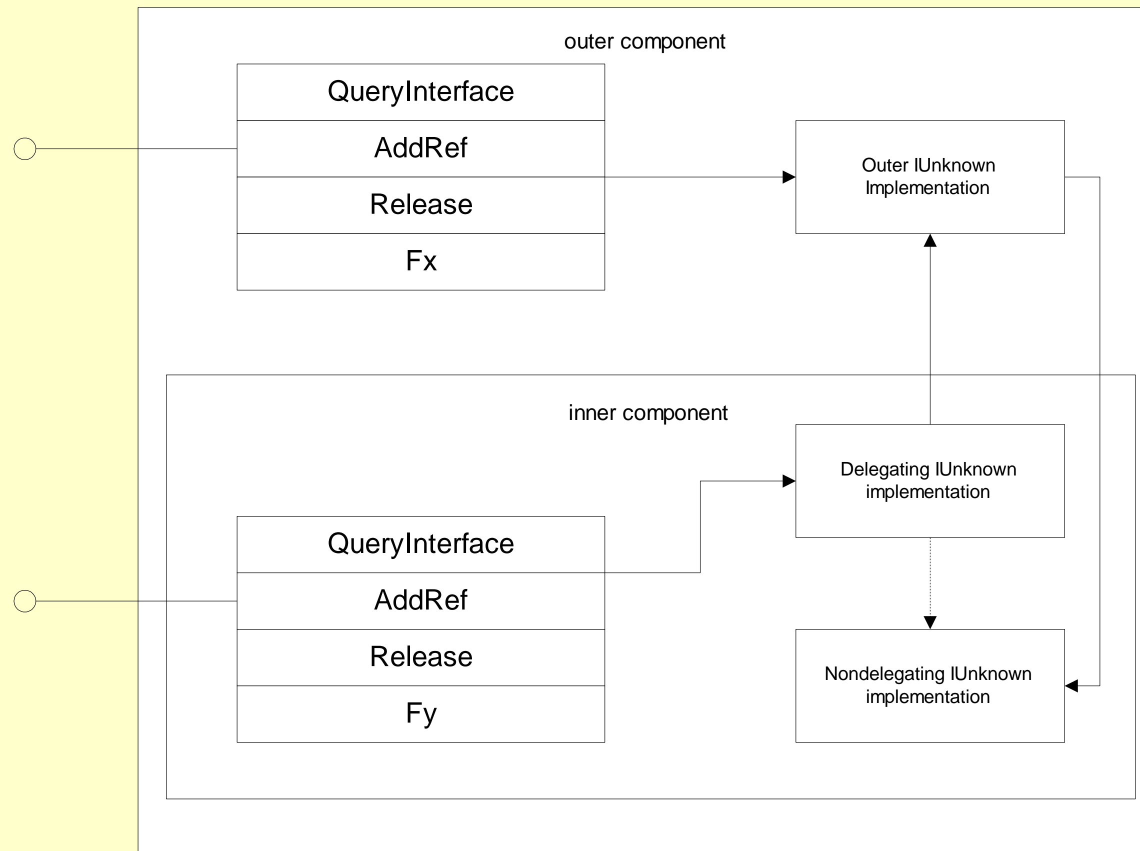
COM Interface Policy







COM Aggregation



Implementing (COM) Aggregation

- Signaling aggregation:
  - CoCreateInstance(…) and IClassFactory::CreateInstance(…) both have a parameter: Iunknown* pUnknownOuter. If this pointer is null the created object will not be aggregated.
  - If An outer component wants to aggregate an inner component it passes its own IUnknown interface pointer to the inner.
- Implementing IUnknown:
  - If an aggregatable component is not being aggregated it uses its non-delegating IUnknown implementation in the usual way.
  - If it is being aggregated it uses its delegating IUnknown to forward requests for IUnknown or outer interface to the outer component. Clients never get a pointer to the inner non-delegating IUnknown. When they ask for IUnknown they get a pointer to the outer IUknown.

Implementing Aggregation

- The delegating IUnknown forwards QueryInterface, AddRef, and Release calls to the outer IUnknown.
- When a client requests an inner interface from an outer interface pointer the outer delegates the query to the inner non-delegating QueryInterface.
- When CoCreateInstance is called by the outer component it passes its IUnknown pointer to the inner and gets back a pointer to the inner IUnknown. This happens in an init( ) function called by the outer's class factory in its CreateInstance function.

COM Architectural Features

- Program to Interfaces - create objects with class factories	- Break compilation dependencies
- implement using dynamic link libraries	- reuse binary code - update without rebuilding
- use registry to locate components - identify components using GUIDS	- clients need no knowledge of where components reside - avoid name clashes with other components
- delegate activation to the OS	- allows components with different threading models to interoperate
- use Remote Procedure Call (RPC) communication and marshalling between processes and machines	- support for distributed architectures, e.g., from OLE linking and embedding to enterprise computing
- use Interface Definition Language (IDL) to describe component's interfaces	- hides some of the ugly code required to handle RPCs