Multi-Threaded Systems with Queues

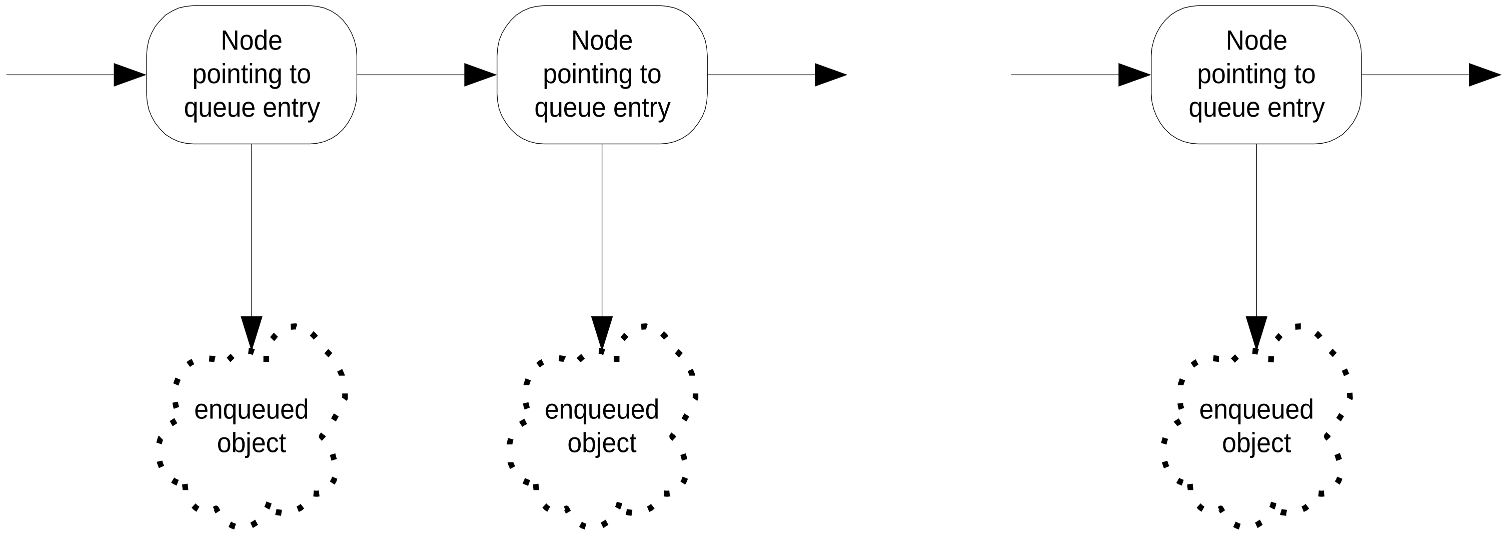
Jim Fawcett

CSE681 – Software Modeling & Analysis

Fall 2003

FIFO Queues

- First In First Out queues are usually constructed with linked lists.
  - Objects enter the queue by getting linked to one end.
  - Objects leave the queue by getting unlinked from the other end.

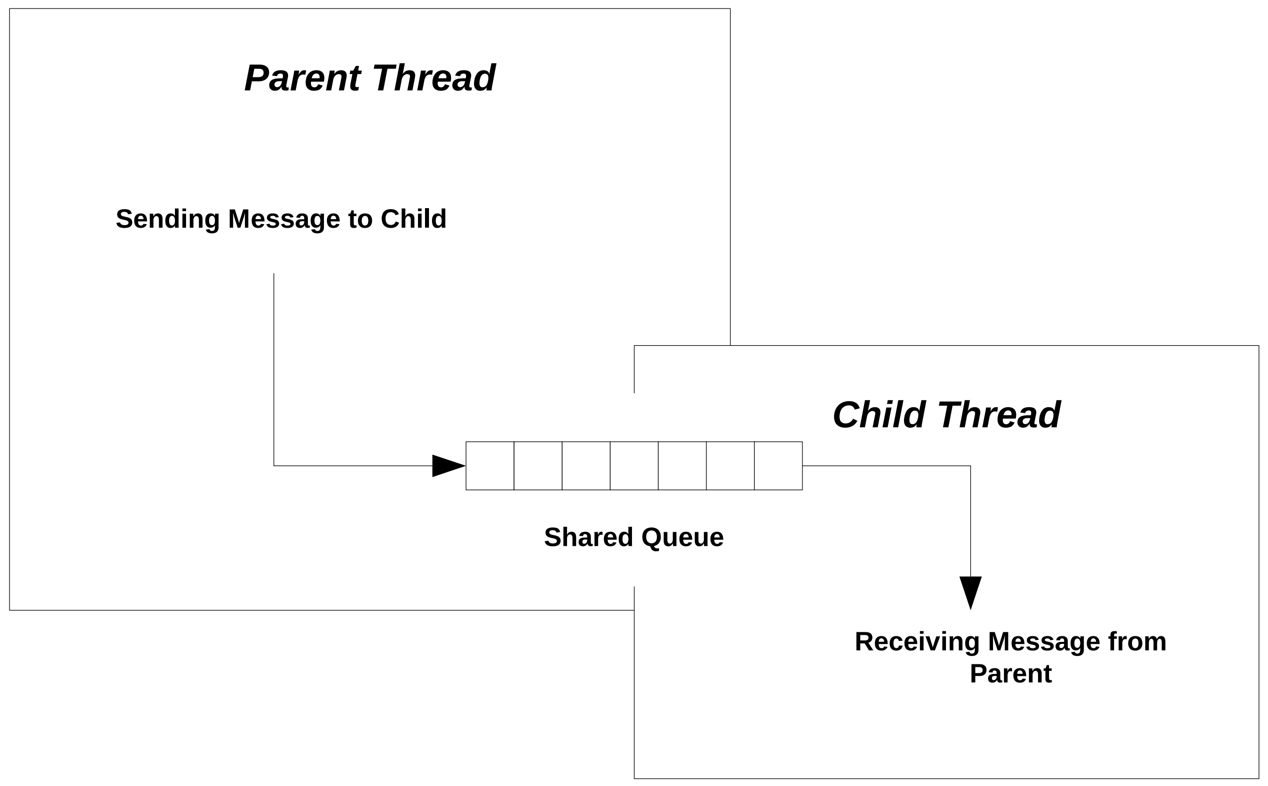


Node pointing to queue entry
Node pointing to queue entry
Node pointing to queue entry
enqueued object
enqueued object
enqueued object

Important Property of Queues

- Queues decouple a receiver from its sender.
  - Sender and receiver can be on different threads of a given process.
  - Receiver does not need to process an object when it is handed off by the sender.
  - Queues can eliminate timing mismatches between sender and receiver.
    - One might be synchronized in nature, requiring message passing at fixed time intervals – a radar signal processor for example.
    - The other might be free-running, handling messages with different service times, preventing synchronized operation.
  - Queues can support reliable communication over unreliable media, simply holding onto messages it can’t send until the transmission link becomes available.

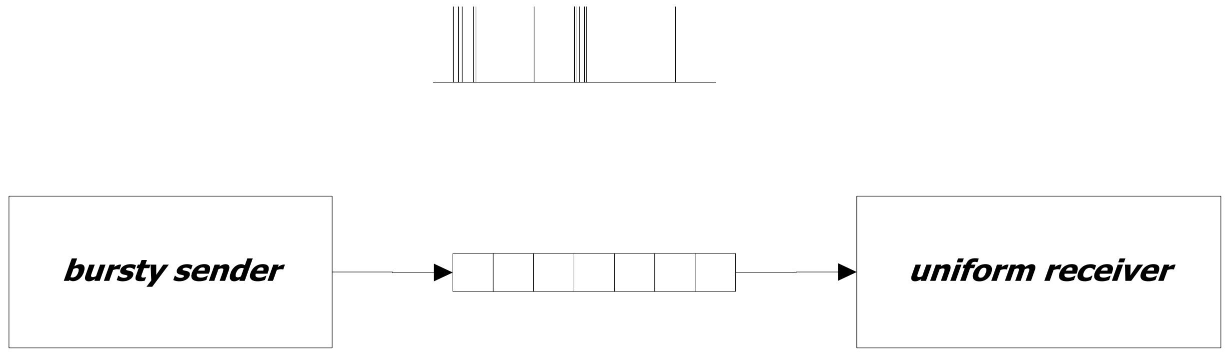
Message Passing Between Threads



Removing Timing Mismatches

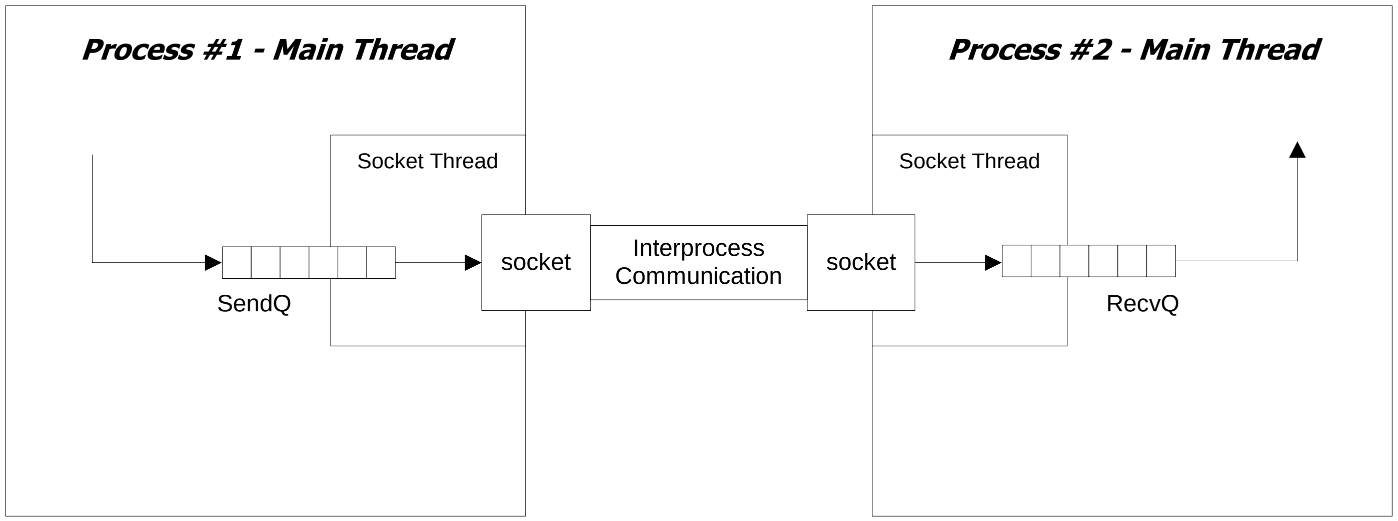
- Radar Signal Processor reports detections at fixed times, but often has no detection to report, so its output is bursty.

Bursty System



Reliable Communication

Store and Forward Architecture

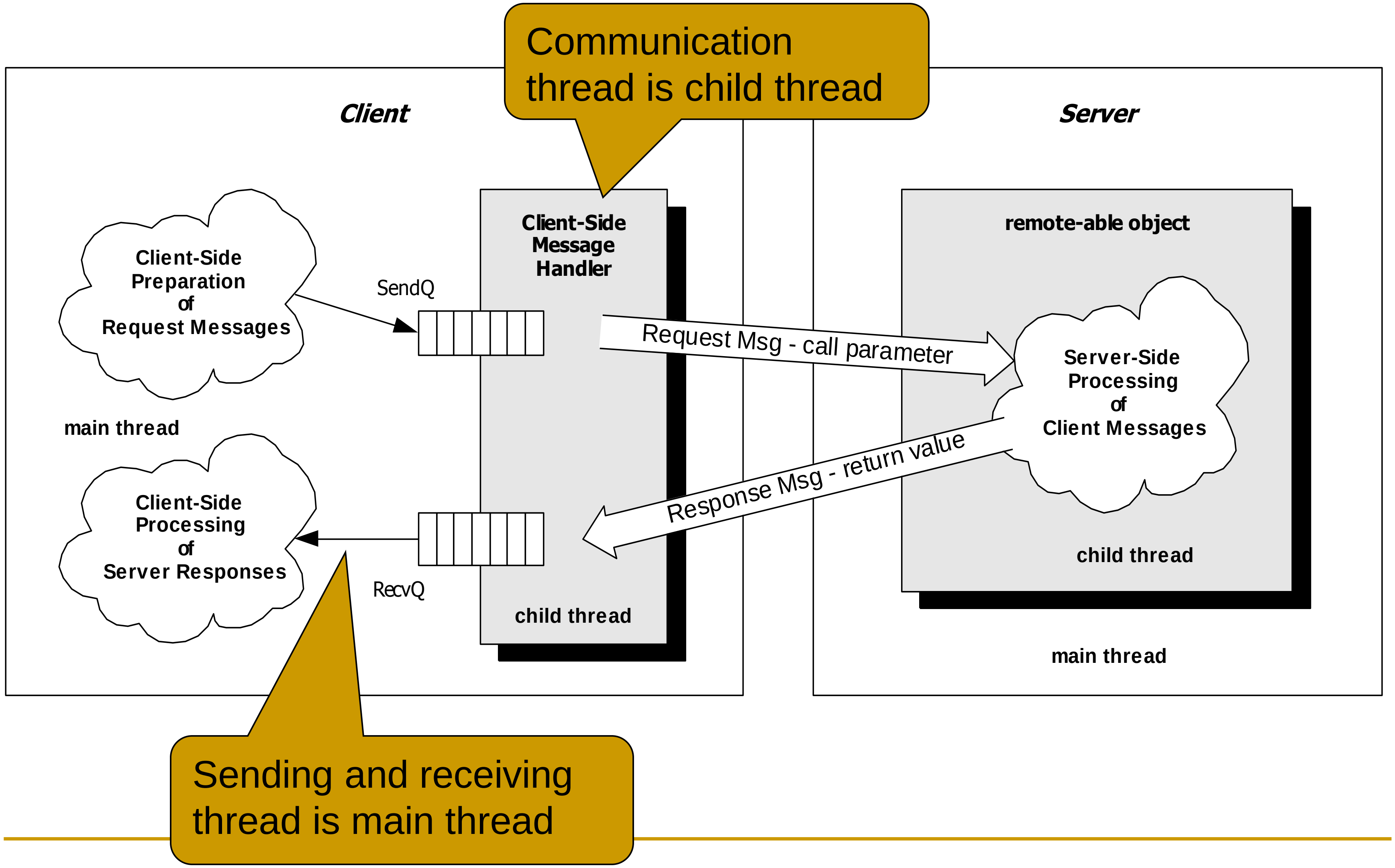


SendQ is used to hold messages in the event that communication with the remote receiver fails.

Messages are held until communication is re-established.

RecvQ is used to quickly remove messages from the socket connection so that the socket buffer never fills (that would block the sender).

Message Passing with Queues via Remoting

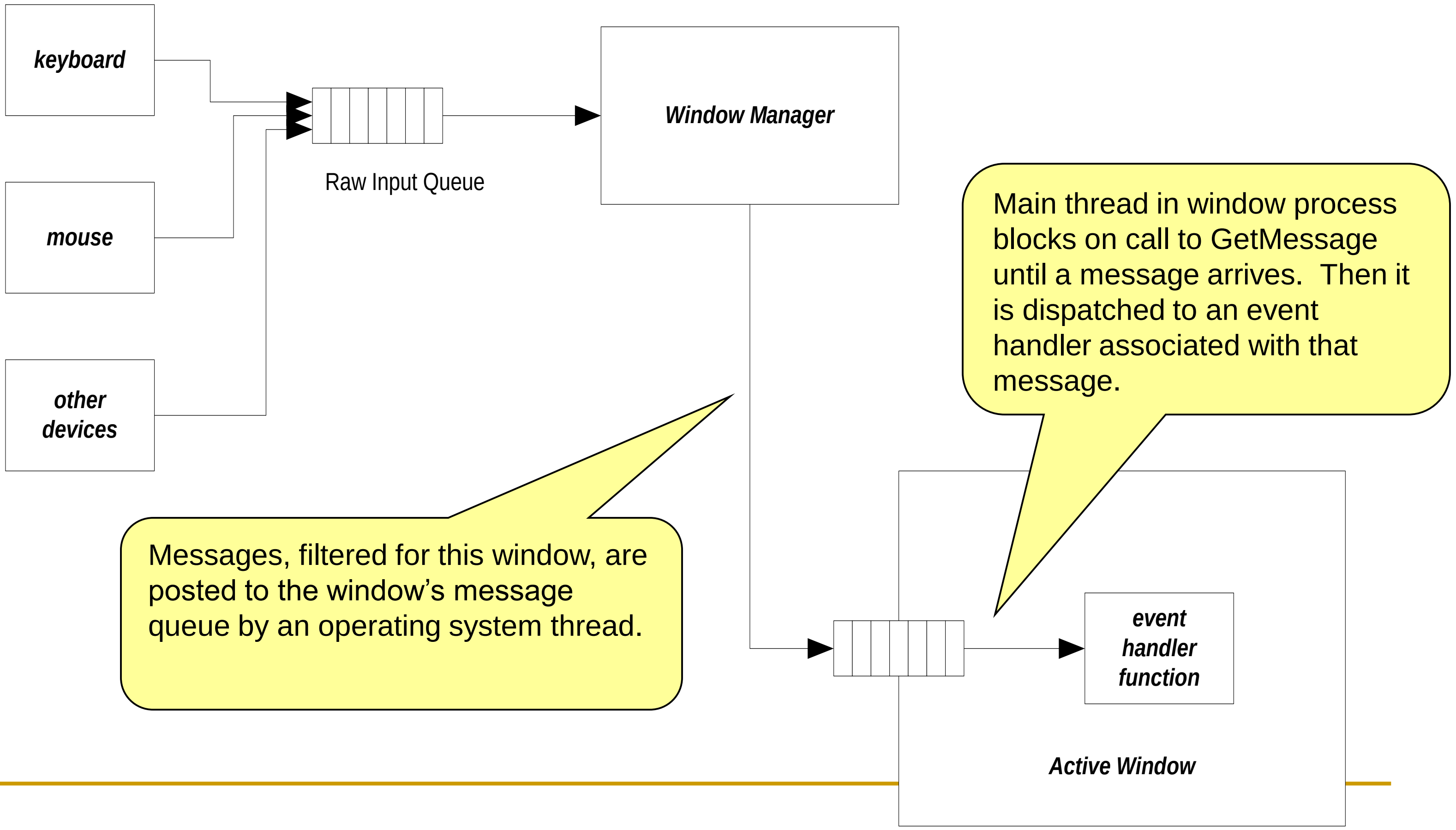

Communication thread is child thread
Client
Server
Client-Side Preparation of Request Messages
SendQ
Client-Side Message Handler
remote-able object
Request Msg - call parameter
Server-Side Processing of Client Messages
main thread
Response Msg - return value
Client-Side Processing of Server Responses
RecvQ
child thread
child thread
main thread
Sending and receiving thread is main thread

Send and Receive Queues

- Essentially, a SendQ lets:
  - Sender thread create a list of requests.
  - Communication thread remember requests it has not processed yet.
- A RecvQ lets:
  - Remote process return a result independently of the receiver’s main thread’s readiness to accept it.
  - Valuable remote resource need not block waiting for a hand-off to receiver.
- Both queues allow the client’s main thread to service other important tasks as well as interact with the remote resource.

Windows, Queues, and Messages

- Graphical User Interfaces are the stereotype of message-passing systems using queues.



Main thread in window process blocks on call to GetMessage until a message arrives. Then it is dispatched to an event handler associated with that message.

Messages, filtered for this window, are posted to the window's message queue by an operating system thread.

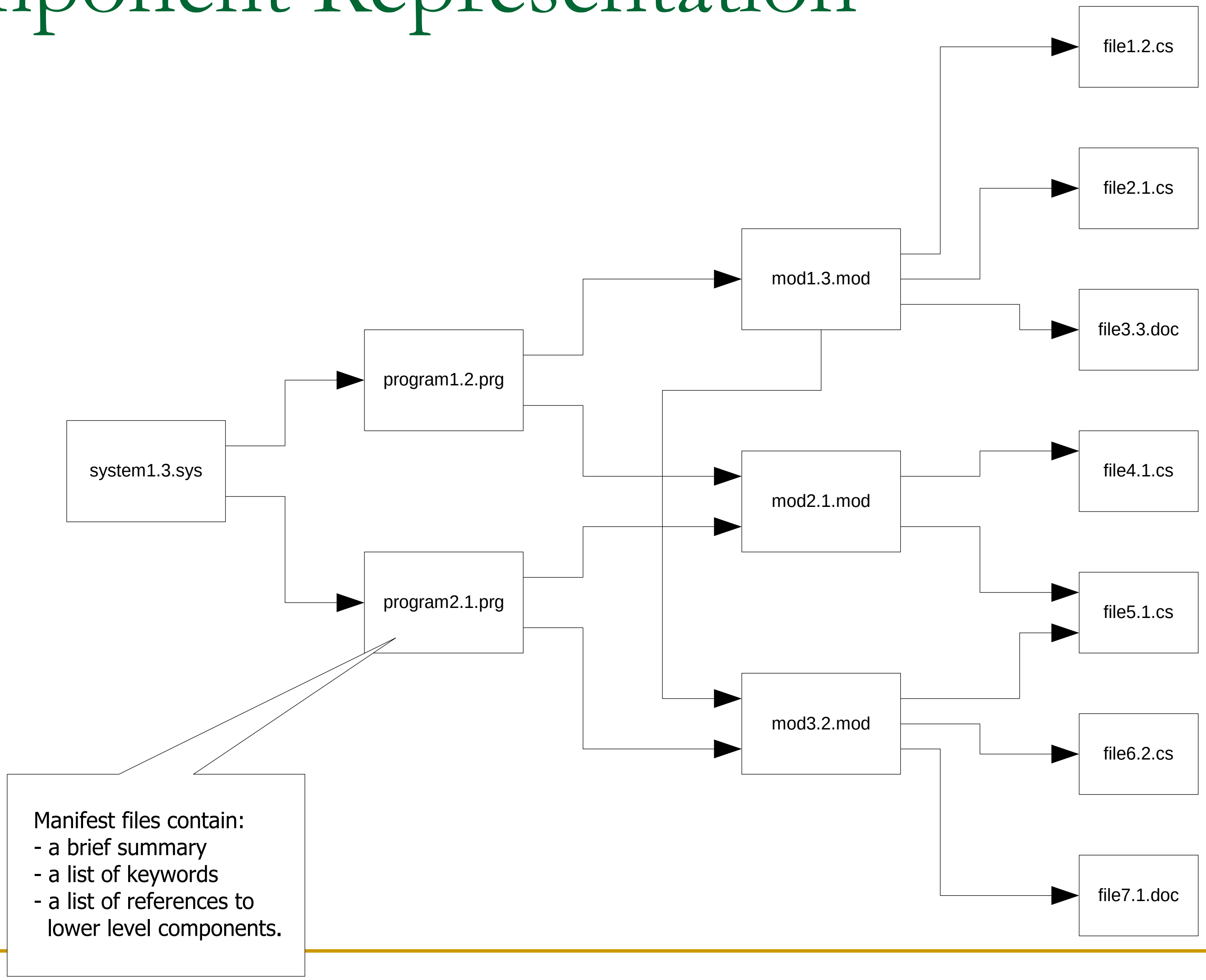
Windows Messaging

- With the architecture shown on the previous slide a window can respond to any of many different inputs, including:
  - User inputs from mouse movement, mouse buttons, and keyboard
  - System events, e.g., being uncovered by an obscuring window, and so needing to repaint its region of the screen
  - Message generated from within the program running in that process, based on strategies put in place by the designer.
- Even if several messages arrive before a predecessor is processed, they won’t be lost.

Component Server Example

- The Component Server of projects #3 and #4, Fall 02, serves sets of files, organized as components.
- A component may be:
  - A single low-level module that links only to files
  - An upper-level module that links to several lower level modules as well as files
    - perhaps test driver and documentation files
  - A program that links to several modules and files
  - A system that links to several programs and files

Component Representation



Manifest files contain:
- a brief summary
- a list of keywords
- a list of references to lower level components.

Component Server Activities

- Show available components
  - Systems or programs or modules
- Make a new component
  - Specify type
  - Show available server-side links
    - If system, then programs and non source files
    - If program then modules and non source files
    - If module then modules and files
  - Specify server-side link
- Extract a component to client
  - Store at specified client path
- Insert a new file in server

Show Available Components

- Either:
  - Ask server to send all its manifests of the specified type
    - Send a command, get back a set of files
- Or:
  - Send a list of manifests available on client
    - Send a command with a list of names (includes version)
  - ask server to send back any manifest client does not have (or later version)
    - Get back a set of files

Make a New Component

- Supply server with name and type
  - Send command with name and type attributes
- Ask server for list of possible targets
  - Send command with type attribute
    - If module:
      - Get back list of most recent version of all modules
      - Get back list of most recent version of all files
    - If program:
      - Get back list of most recent version of all modules
      - Get back list of most recent version of all non-source files
    - If system:
      - Get back list of most recent version of all programs
      - Get back list of most recent version of all non-source files
- Specify links
  - Send server command with component name and names of links

Extract a Component

- Specify target path
  - Tell client file handler where to put files
- Supply server with name
  - Example:
    <extract myProg.2.prg>
  - Get back set of files

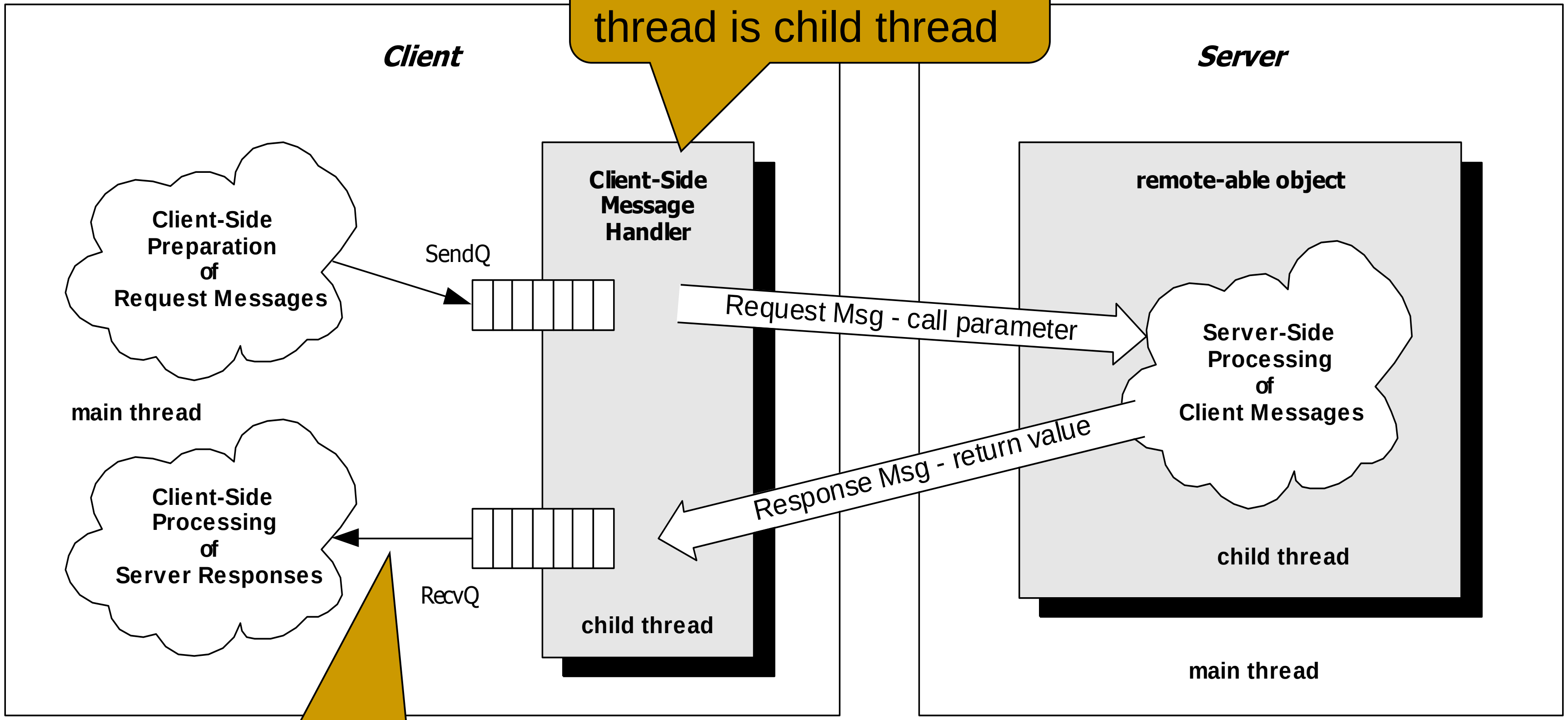
Issues with Component Server

- Many of these operations may take a significant amount of time.
- Client may have other important things to do:
  - Probably has a user interface to support:
    - Needs to service message queue at regular intervals.
  - May want to spawn compiles on received files
  - Support user browsing of manifest links and file text
- Therefore, we will want to make server requests on a worker thread.
  - This allows client activities to progress concurrently with communication to the server.

More Issues

- Since operations are initiated by a user through GUI they arrive asynchronously.
- Furthermore, requests may be bunched or may be made widely separated in time.
- User actions may attempt to initiate new operations before previous requests have been completed.
- One effective way to handle this situation is to use queues, as shown in the next slide.

Message Passing with Queues


Client
Server
Client-Side Message Handler
remote-able object
Client-Side Preparation of Request Messages
SendQ
Request Msg - call parameter
Server-Side Processing of Client Messages
main thread
Response Msg - return value
Client-Side Processing of Server Responses
RecvQ
child thread
child thread
main thread

Communication thread is child thread

Sending and receiving thread is main thread

Issues with Message Passing

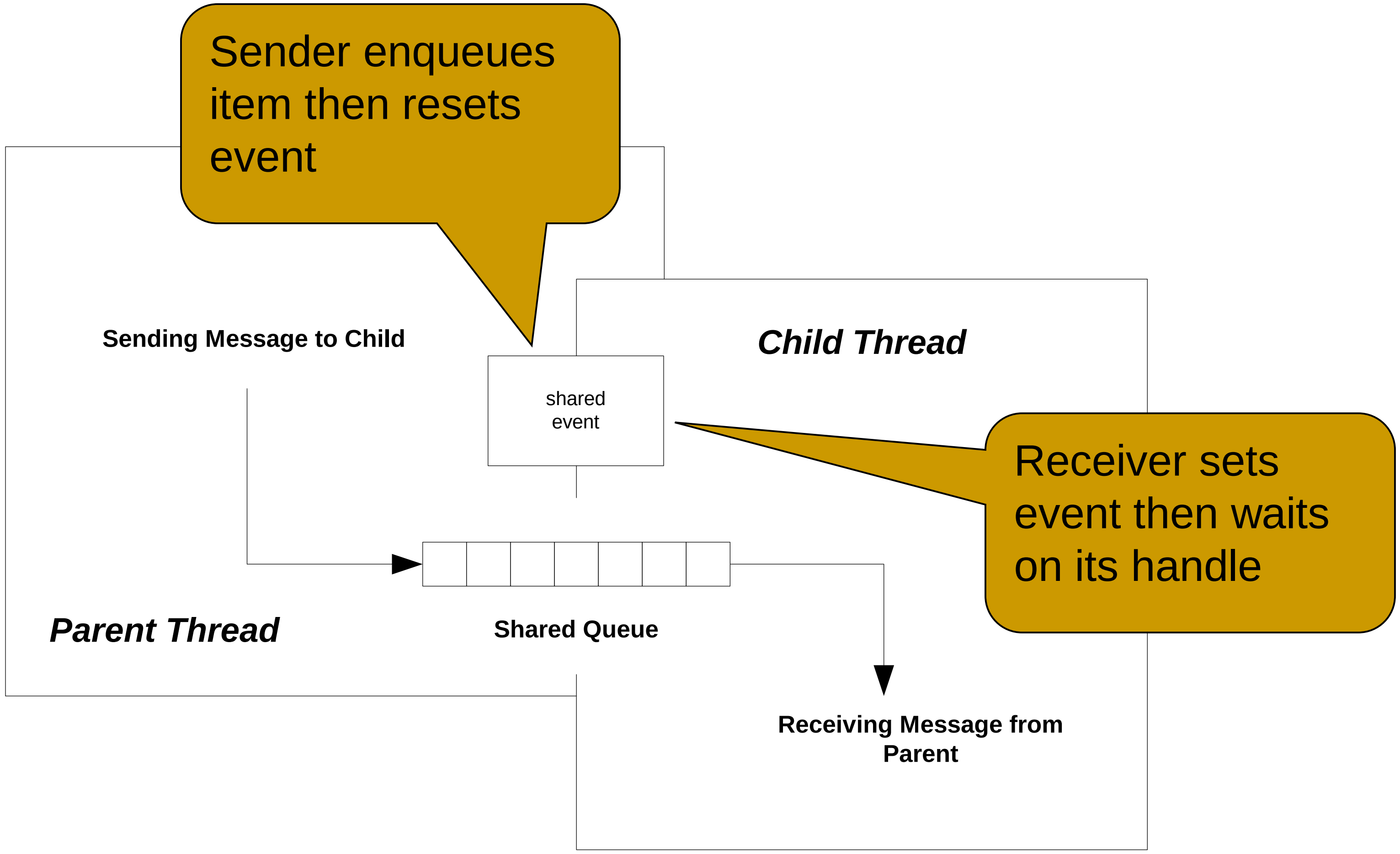
- The main question with this approach is:
  - how does the receiving thread know when to check its RecvQ?
- If the only thing the receiving thread does is process its messages – like windows – it doesn't need to know.
  - It simply blocks on the queue until something becomes available.
- However, for systems like the Component Server, that have several things to do, there are better solutions:
  1. The receiver's main thread can poll the RecvQ periodically and then go about its other business.
  2. The receiver's communication thread can notify its main thread when there is something to service in the queue.

  We will look at all these solutions in the next few slides.

Blocking Call

- Suppose that the only thing receiver does is process messages from RecvQ.
- Main thread can sleep for a some Milliseconds then check the queue. This wastes CPU cycles if the queue is often empty.
- How does a receiver block until queue has a message?
  - Sending thread and receiving thread share an event.
  - If queue is empty when receiver checks, it sets the event and waits on the event handle.
  - When the sender posts a message it fires the event, waking the receiver to process the message.

Blocking Call (continued)


Sender enqueues item then resets event
Sending Message to Child
Child Thread
shared event
Receiver sets event then waits on its handle
Parent Thread
Shared Queue
Receiving Message from Parent

Blocking Dequeue

- Example code in /handouts/CSE681/code/BlockingQueueVS2003.
- Note caveat concerning single writer/single reader.

Polling RecvQ

- If the receiving thread is a UI thread it can:
  - Create a timer, set to tick at the polling time interval.
  - Define a message handler that checks the queue and removes available entries.
  - This works as follows:
    - When the timer is started it sends a timer message to the windows message queue of the UI thread.
    - It sends the message again, every time the timer ticks.
    - We simply associate the timer with its event hander, OnTick, through the system defined EventHandler:

```
timer.Tick += new EventHandler(OnTick);
```

    - When the UI thread is not handling timer Tick events it can handle any other UI events that occur.

Polling Dequeue

- Example code would be useful here.
- None provided yet.

Notifications

- If the receiving thread is a UI thread it can:
  - Define an event handler to dequeue messages from the RecvQ.
- The communication thread defines a delegate and points to the event handler.
- When communication thread returns from remote call it:
  - Enqueues the reply message
  - Calls Form.beginInvoke(delegateName, object[] { args…}) on the delegate.
  - This marshals call to the UI thread causing the reply message to be processed on the main thread.

Notifications Dequeue

- Example code would be useful here.
- None completed yet.

Non UI Receiver

- If the receiver thread is not a UI thread it is very likely that the only thing it does is process messages from the queue.
  - In this case, a blocking call is exactly what is needed.
- If we have the rare situation where this is not the case, all is not lost. We can create a hidden window to process:
  - Notifications from the communication thread
  - Notifications from other theads needed to do the rest of the receiver's business.
  - Since the window is hidden it won't have any user actions to process.

Client with Three Threads

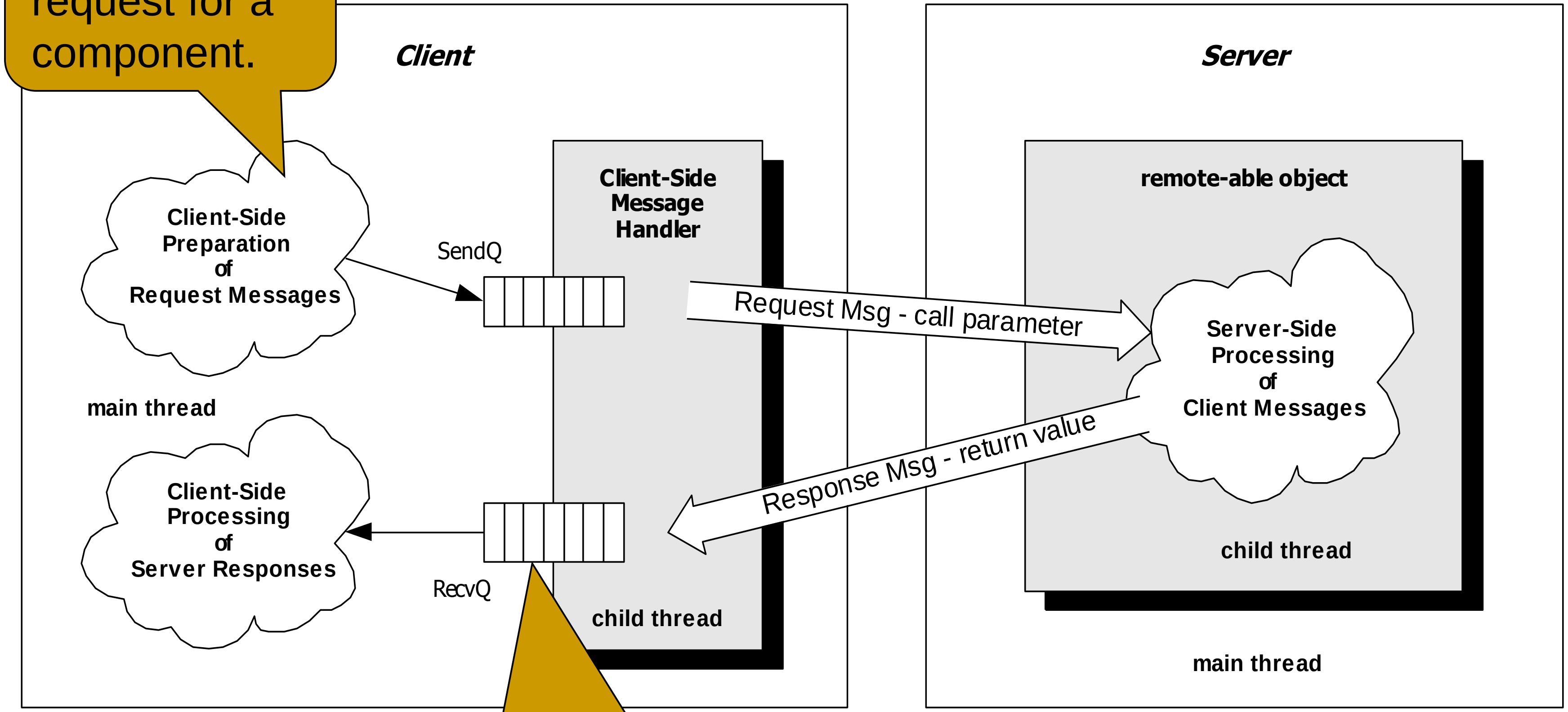
- One design that I have used and liked uses three threads:
  - The main thread handles user events like mouse actions and key presses and enqueues requests for the server.
  - A communication thread takes care of interacting with the server as shown previously.
  - A receive thread is responsible for dequeuing messages posted by the communication thread and populating various controls in the UI, using BeginInvoke to marshal calls to the main thread.
- This design makes the receive thread block on an empty RecvQ.
- The protocol is simple.

File Serving

- Many of the messages sent to the component server ask it to send back some collection of files – those belonging to some component.
- As the files reach the client, the communications thread will want to save them in some specified path, then post the names in the RecvQ. This way the UI can display the names as they are received, allowing the client user to access them only when they have fully arrived.
    - Perhaps a click on a filename in a listBox will display the file text in a window.
    - We don’t want the name to appear in the listBox until the file has arrived.

Message Passing with Queues

Client creates request for a component.


Client
Server
Client-Side Preparation of Request Messages
main thread
Client-Side Processing of Server Responses
SendQ
RecvQ
Client-Side Message Handler
child thread
Request Msg - call parameter
Response Msg - return value
remote-able object
Server-Side Processing of Client Messages
child thread
main thread

Child thread saves files and posts names as they arrive.

File Serving Issue

- There is another interesting file serving issue:
  - Small text files can be returned as a single string.
  - Bigger files and all binary files will need to be return as blocks, with each block consisting of an array of bytes – remoting handles that almost as easily as passing strings.
  - However, if we use a singlecall object, we can only get a single block back before the object will be destroyed, and the file pointer will be discarded with it.
- There are three fairly straight-forward solutions to this problem:
  1. Use a singleton object and provide synchronization for concurrent clients – this is fairly straight-forward and won’t be discussed further.
  2. Pass a file saver object by reference to the remote object.
  3. Create a client activated remote object.

Types of Remoting Servers

- Server Activated
  - SingleCall
    - Object is created by server when a call arrives.
    - Object is destroyed as soon as the call completes.
    - Each client gets a unique instance.
    - Established by RegisterWellKnowServiceType with parameter WellKnownObjectMode.SingleCall.
  - Singleton
    - Object is created by server when first call arrives.
    - Object is destroyed when its lease expires.
    - Each client gets reference to singleton.
    - Established by RegisterWellKnowServiceType with parameter WellKnownObjectMode.Singleton.
- Client Activated
    - Actually still activated by server when client calls new. Unlike Server Activated constructor may have parameters.
    - Object is destroyed when its lease expires.
    - Each client gets a unique instance.
    - Established by RegisterActivatedServiceType instead of RegisterWellKnowServiceType call.

Passing Arguments by Reference

- File saver object:
  1. Pass a file saver object by reference as a parameter to remote call.
  2. Server calls a member function to accept a block, then calls another function asking object to save the block.
  3. Server repeats this process until entire file has been sent, then returns.
- Since file saver object is passed by reference, when the server calls its member functions the data is marshaled back to the client side where the real object resides.

Passing File Saver Object by Reference

- You will find a Pass-By-Reference example in the folder PassByRef.
- It does not, of course, implement a file saver object. That is up to you.
- Note: As of VS2003 pass-by-reference objects throw security exceptions when server makes calls back to client using the reference. I expect this can be addressed by managing security settings, e.g., make the server a trusted source and set the trusted zone permissions accordingly. I have not had time to check that out yet.

Client Activated Object

- Client activated objects continue to exist after a client’s call.
  - The remote object’s lifetime is determined by a lease timeout, and can be programmatically set.
  - Thus, as long as the timeout has not expired, the client may keep calling the object until all blocks have been sent.
  - The server can notify the client’s communication thread simply by returning a boolean value. True means there is more to send.

Thoughts about Project #4 Interface

- Project #4's tasks are:
  - Extract and View components held by a remote resource.
    - Please make sure that this also works with localhost since I will test most of the projects that way.
  - Create a new component
  - Insert new files into the component server
    - These will be used to make some of the new components and add to others.
- Because of time limitations you will want to make the user interface simple.
  - It still must be effective for the tasks listed above.

End of Presentation