
SYSC 3303 Real-Time Concurrent Systems

TFTP Client-Server Design Part 1

- Copyright © 2000-2003 D.L. Bailey, Copyright © 2002-2003 L.S. Marshall, Systems and Computer Engineering, Carleton University
- revised February 10th, 2003

About These Slides

- In this set of slides, we'll use UCMs to design a client-server system that implements TFTP
- Initially, we'll assume that packets are never lost, delayed or duplicated in the network

Constraints for Version 1 of the System

- A TFTP client cannot transfer multiple files simultaneously; i.e., it does not support concurrent read or write connections
- A TFTP server handles one connection at a time; e.g., while sending/receiving a file to/from one client, it ignores connection requests from other clients

Scenarios

- 4 major scenarios can be identified from the TFTP spec.:
 - establish WRQ connection
 - write file from client to server (successful establishment of a WRQ connection is a prerequisite)
 - establish RRQ connection
 - read file from server to client (successful establishment of a RRQ connection is a prerequisite)

Establish WRQ Connection: Client Responsibilities

- form-WRQ: prepare a TFTP WRQ packet
- form-dgram: prepare a UDP datagram containing the TFTP WRQ packet
- send-dgram: send the UDP datagram (destination is port 69 on the server's host)
- rcv-dgram: receive a UDP datagram
- extract-msg: extract the message from the UDP datagram
- verify-ACK: verify that the received message is a valid TFTP ACK packet

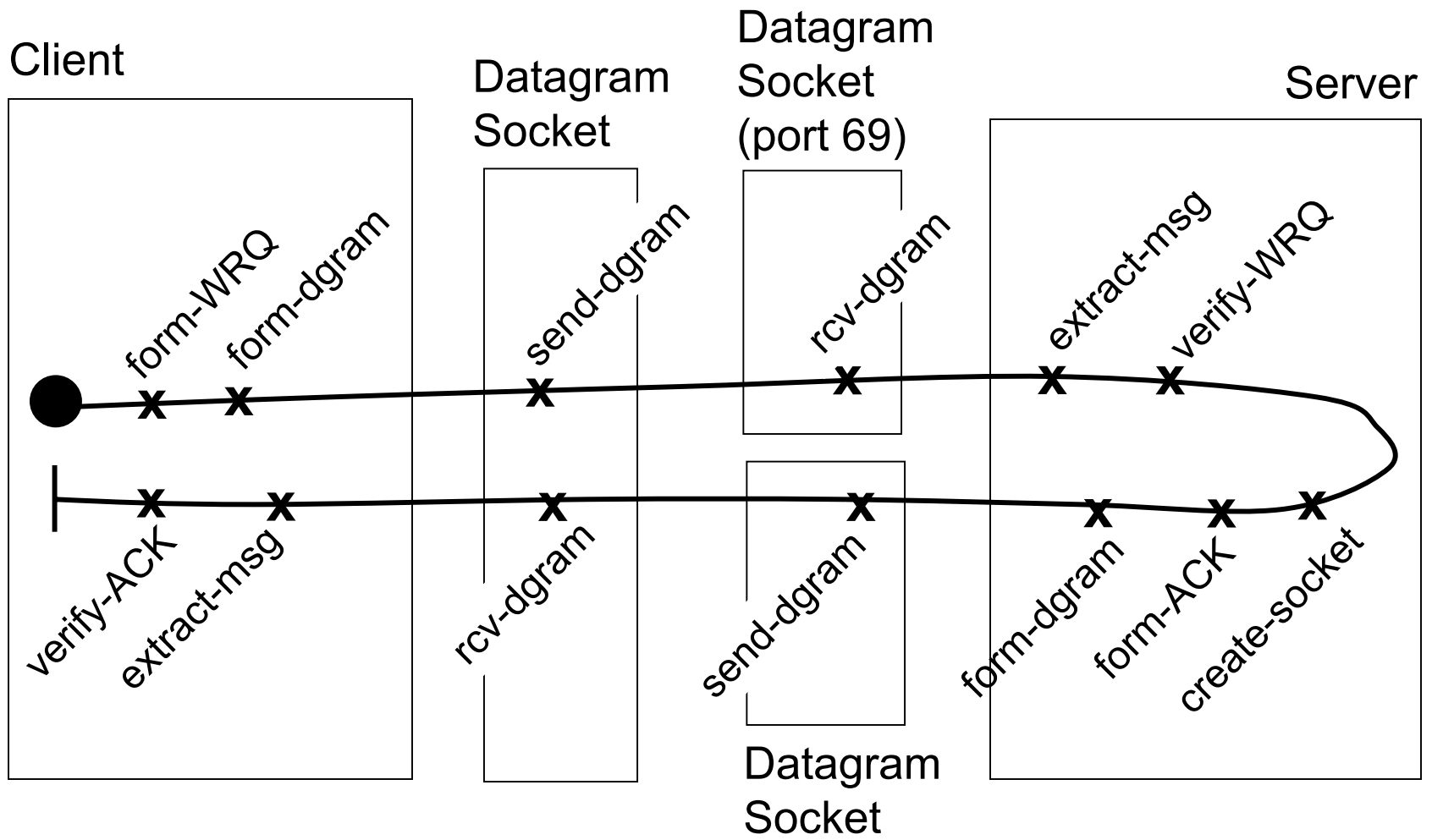
Establish WRQ Connection: Server Responsibilities

- rcv-dgram: receive a UDP datagram
- extract-msg: extract the message from the UDP datagram
- verify-WRQ: verify that the received message is a valid TFTP WRQ packet
- create-socket: create a datagram socket to send TFTP ACK packets and receive TFTP Data packets
- form-ACK: prepare a TFTP ACK packet, with block number = 0
 - cont'd on next slide...

Establish WRQ Connection: Server Responsibilities

- form-dgram: prepare a UDP datagram containing the TFTP ACK packet
- send-dgram: send the UDP datagram (destination is the client's datagram socket)

UCM for Establishing a WRQ Connection



Things to Note

- We've shown only the key responsibilities
 - if we need to show more, draw a larger UCM, or use *layering* (not covered in this course, but see the UCM paper on the course Web site)
- We've not yet identified the concurrency within the system and decided on the number of concurrent threads required

Transferring the File: Client Responsibilities

- form-DATA: read up to 512 bytes from the file, store them in a new TFTP DATA packet
- form-dgram: prepare a UDP datagram packet containing the TFTP DATA packet.
- send-dgram: send the UDP datagram (destination is the datagram socket created by the server during connection establishment)
- rcv-dgram: receive a UDP datagram
- extract-msg: extract the message from the UDP datagram
- verify-ACK: verify that the received message is a valid TFTP ACK packet

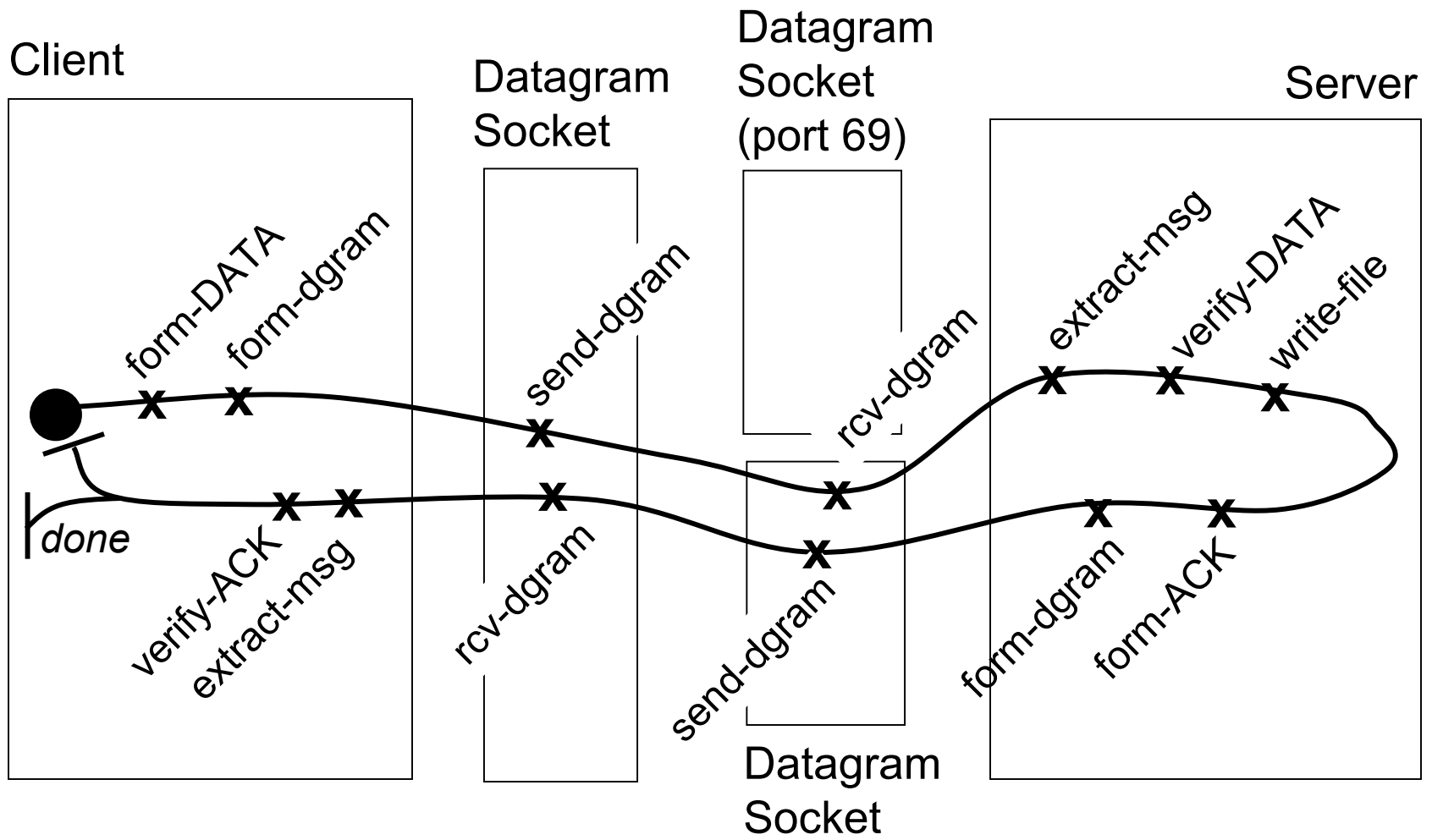
Transferring the File: Server Responsibilities

- rcv-dgram: receive a UDP datagram
- extract-msg: extract the message from the UDP datagram
- verify-DATA: verify that the received message is a valid TFTP DATA packet
- write-file: extract the data from the TFTP DATA packet and write it to the file
- form-ACK: prepare a TFTP ACK packet
 - cont'd on next slide...

Transferring the File: Server Responsibilities

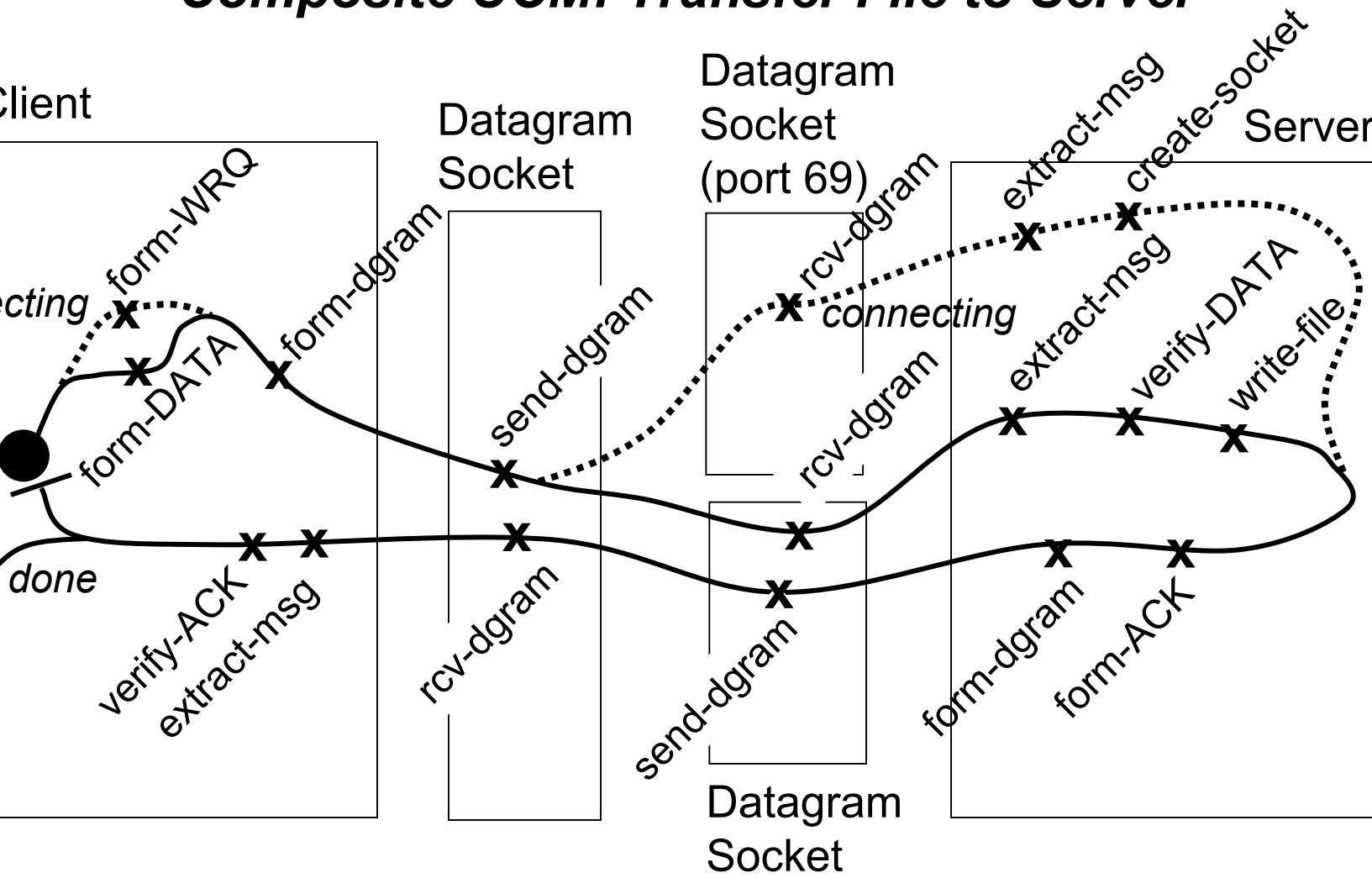
- form-dgram: prepare a UDP datagram containing the TFTP ACK packet
- send-dgram: send the UDP datagram to the client's datagram socket

UCM for Steady-State File Transfer to Server



Things to Note

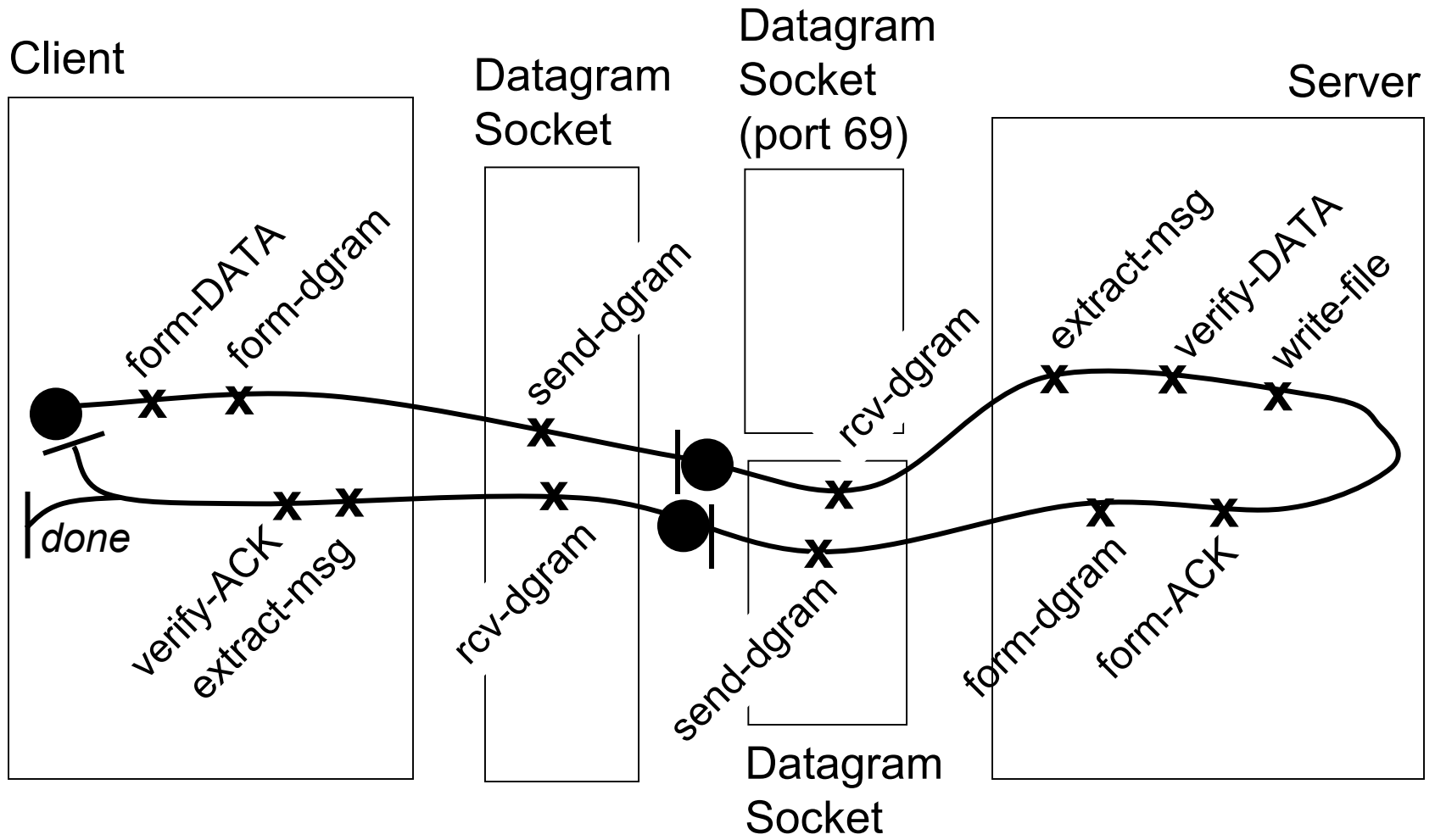
- Completion of one end-to-end traversal of the path (i.e., one block of the file transferred) triggers the start of the next traversal
 - shown by placing the end point adjacent to the start point
- Label *done* adjacent to one of the path segments leaving the OR-fork indicates that this is the segment that is traversed when file transfer is complete
 - OR-fork does not have decision logic at the UCM level of abstraction
 - the composite path results from the superposition of two paths



Discovering Threads

- Discovering concurrent processes/tasks/threads is the most difficult part of real-time systems design for novices (and many experienced designers!)
- Our approach: *let the paths tell you about the processes*
- We'll begin by *factoring* the map for steady-state file transfer into simpler local maps
- We should also look at the map for establishing the WRQ connection, but there is likely to be more concurrency in the steady-state file transfer, so we'll start with that map

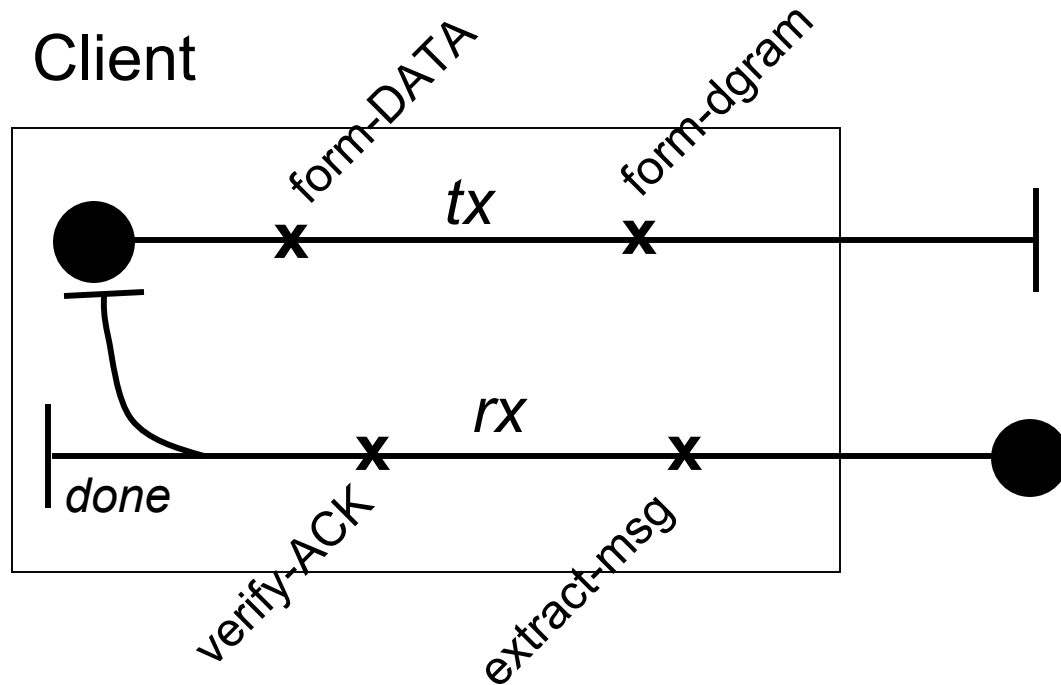
Factoring the Map



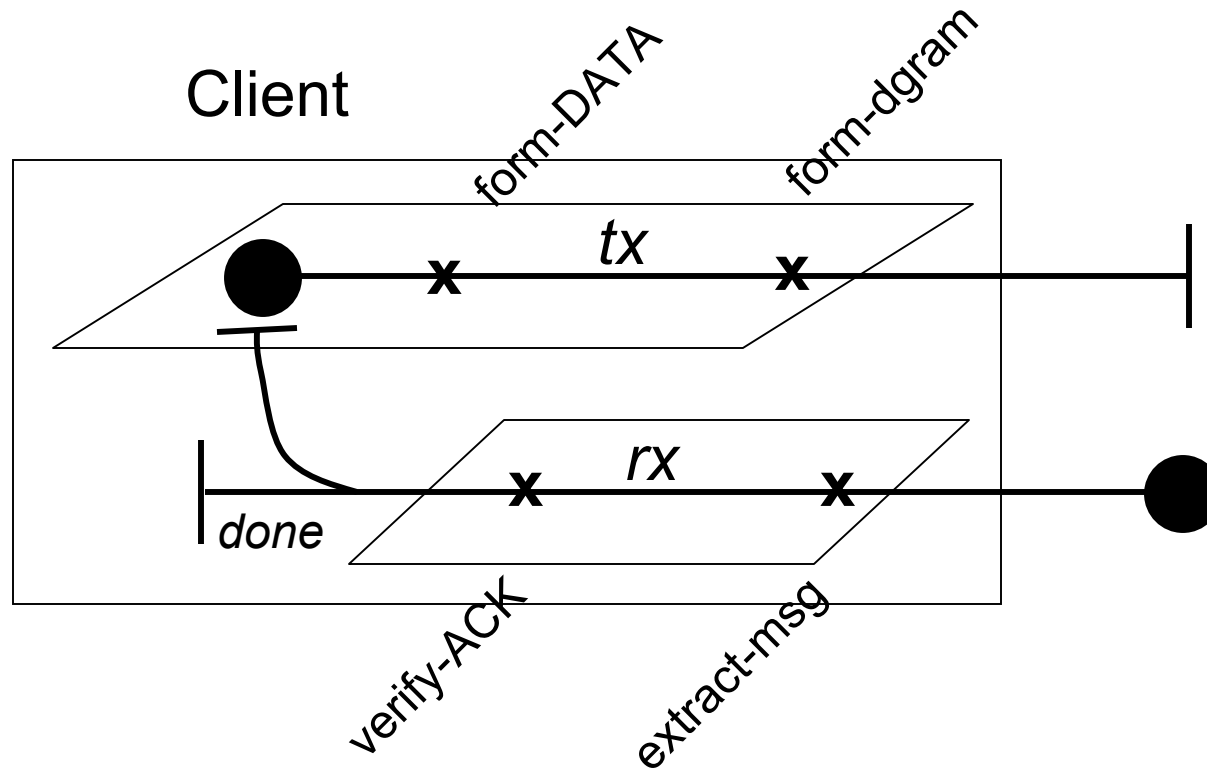
Reasoning About Concurrency

- We're now ready to look more closely at the concurrency in the problem space (as expressed by the paths) and map it to concurrency in the solution space (threads and thread architectures)
- The datagram sockets are 3rd-party components, and we have no control over the concurrency inside them
 - therefore, we won't show the sockets on the following slides

Reasoning About Concurrency in the Client



Candidate Thread Architecture: Equal Peers

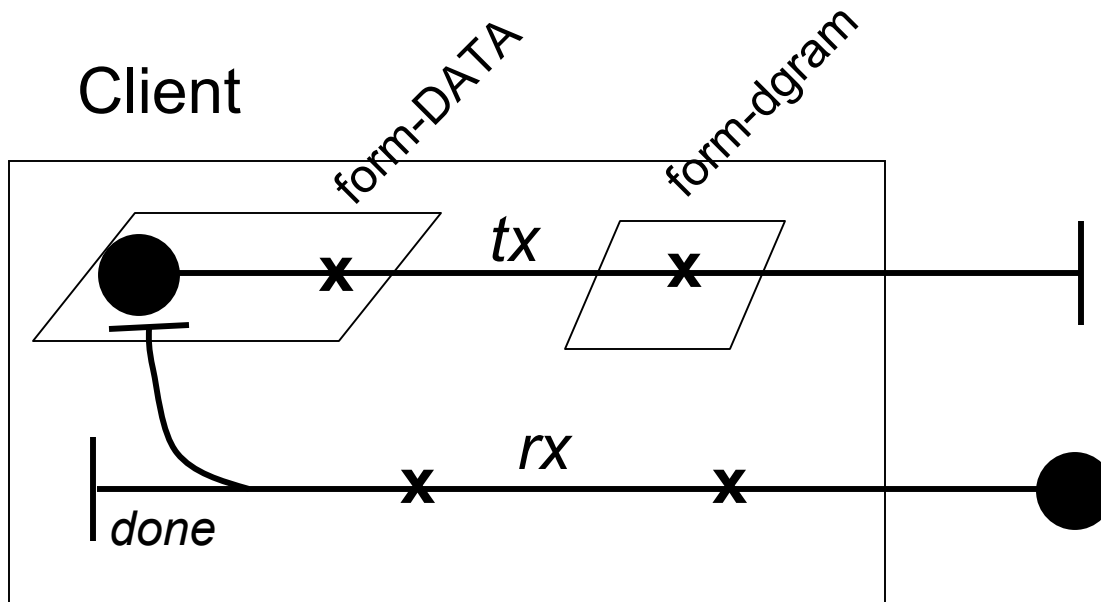


- Allocate the *tx* path to one thread and the *rx* path to another thread?
 - cont'd on next slide

Candidate Thread Architecture: Equal Peers

- Are the *tx* and *rx* paths concurrent relative to one another?
- No (due to lock-step nature of the protocol)
- So, allocating a thread to each path does not increase the system concurrency

Candidate Thread Architecture: Parallel Pipeline



- Allocate each responsibility on the *tx* path to a separate thread?
 - cont'd on next slide

Candidate Thread Architecture: Parallel Pipeline

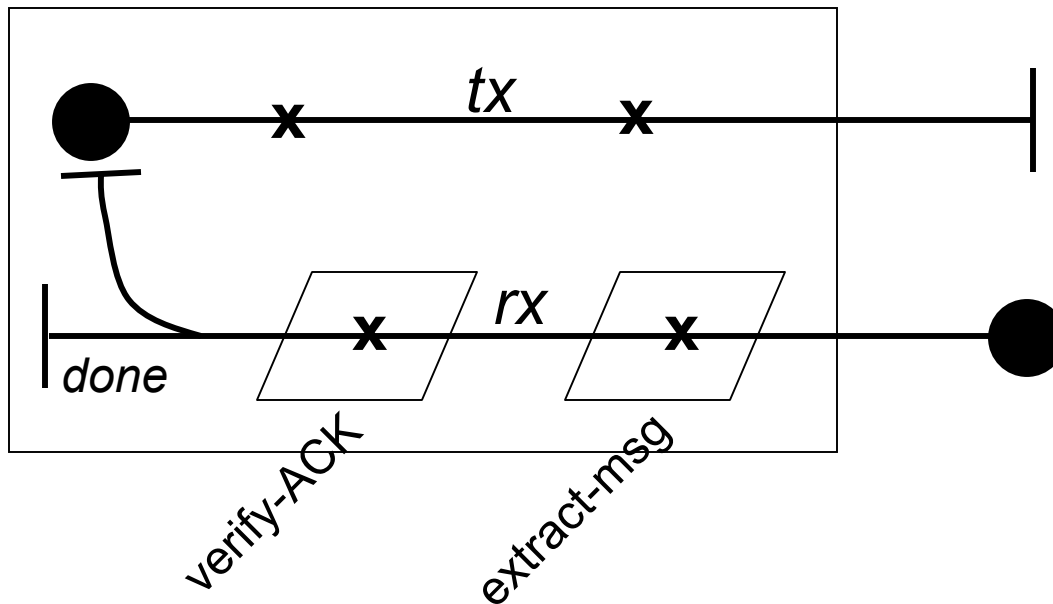
- This architecture is intended to allow progress along the *tx* path to be concurrent
 - one thread forms a UDP datagram while another thread forms the TFTP DATA packet for the next datagram
- But, how much concurrency is there along the path?
 - a new TFTP DATA packet & UDP datagram are prepared only in response to receiving an ACK that the previously transmitted packet was received

Candidate Thread Architecture: Parallel Pipeline

- The lock-step nature of the protocol means that there is no practical benefit to this architecture
- Threads are not a good way to achieve code modularity

Candidate Thread Architecture: Parallel Pipeline

Client



- Allocate each responsibility on the *rx* path to a separate thread?
 - cont'd on next slide

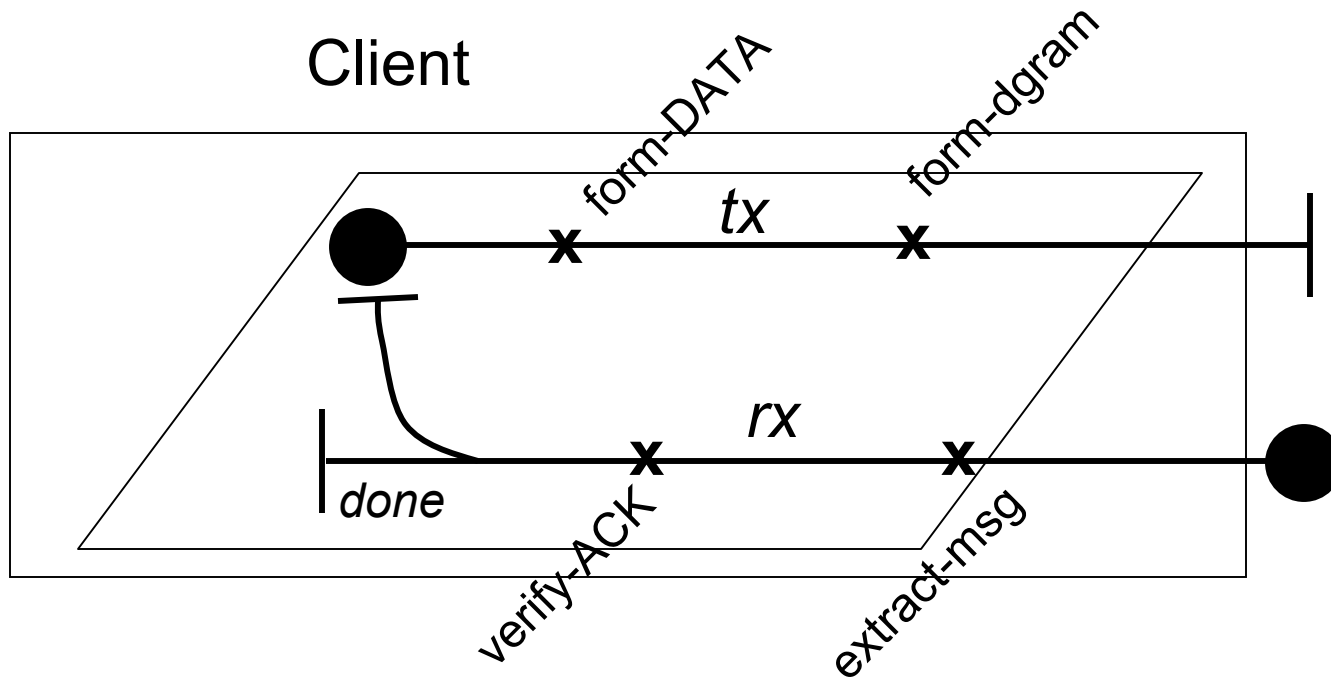
Candidate Thread Architecture: Parallel Pipeline

- This architecture is intended to allow progress along the *rx* path to be concurrent
 - one thread extracts a TFTP packet from a UDP datagram while another thread verifies that packet is the expected TFTP ACK packet
- But, how much concurrency is there along the path?
 - a new UDP datagram containing a TFTP ACK packet should never arrive before the client has finished handling the most recently received ACK, so the client will never perform the `extract-msg` and `verify-ACK` responsibilities concurrently

Candidate Thread Architecture: Parallel Pipeline

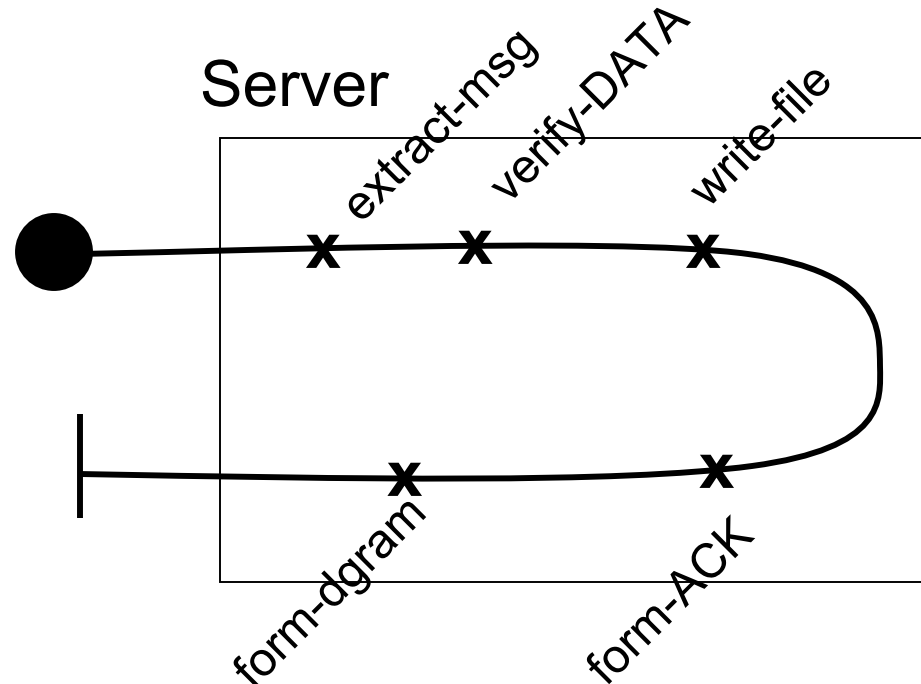
- The lock-step nature of the protocol means that there is no practical benefit to this architecture
- Again, threads are not a good way to achieve code modularity

Candidate Thread Architecture: Singleton



- All responsibilities allocated to a single thread
 - not a good architecture if there is concurrency between or along paths, but looks o.k. here

Reasoning About Concurrency in the Server



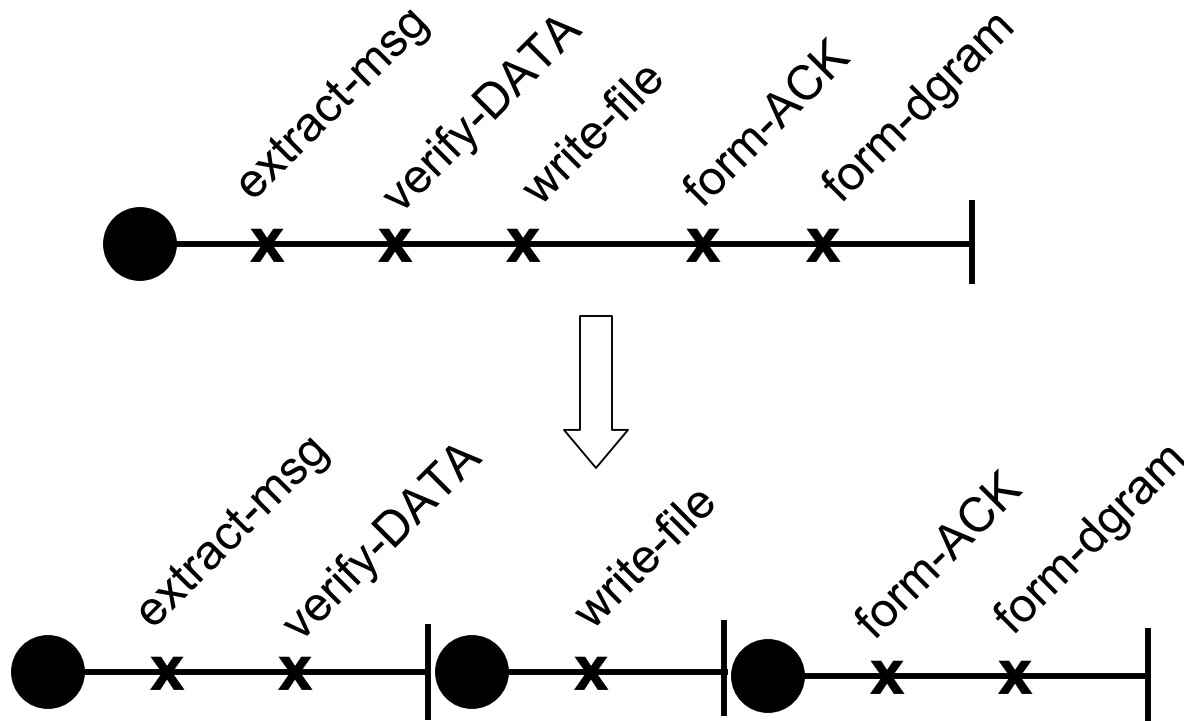
- The lock-step nature of the protocol rules out a pipeline of threads
- A single thread would suffice, but let's look at the responsibilities more closely

Reasoning About Concurrency in the Server

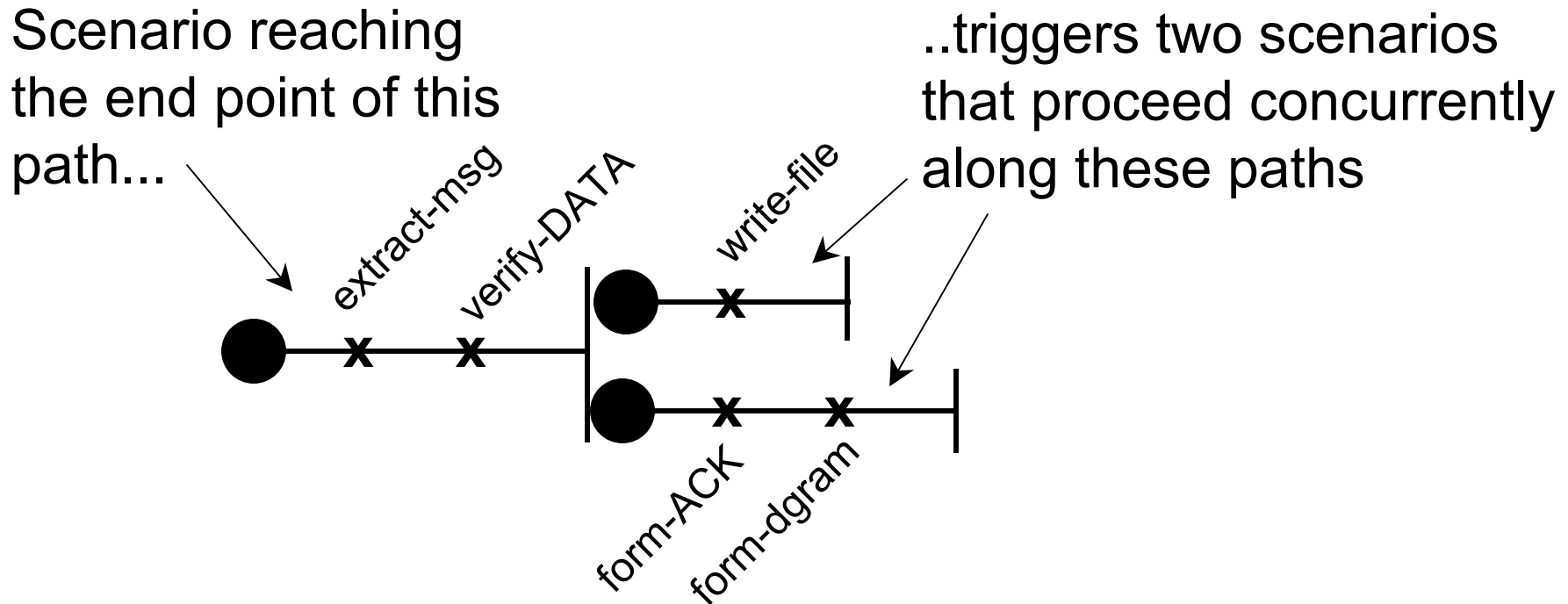
- Responsibilities `extract-msg` and `verify-DATA` must be completed, in that order, before other responsibilities
- Responsibilities `form-ACK` and `form-dgram` must be performed, in that order, after `verify-DATA` is finished
- What about `write-file` and `form-ACK`?
 - we could swap the order in which these responsibilities are performed
 - as such, they can be performed concurrently

Adding Concurrency to the Server Map

- Factor the map:

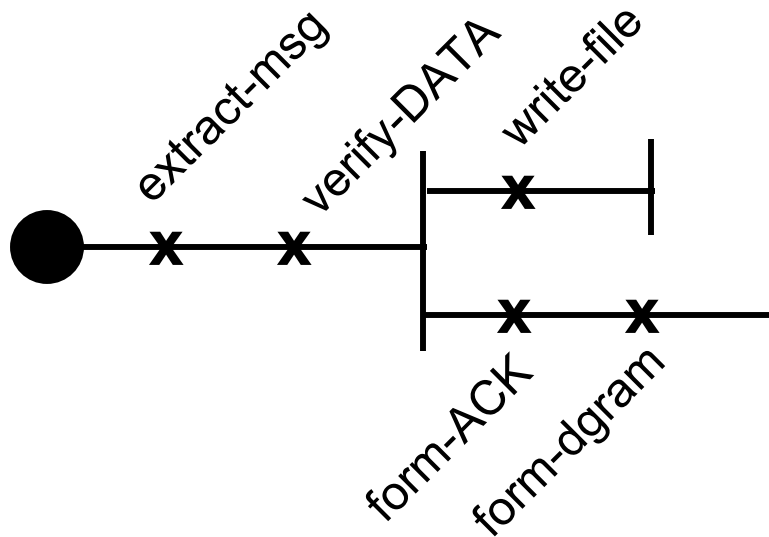


- Rearrange paths to indicate concurrency:

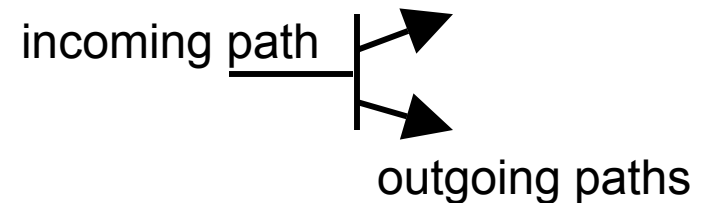


Adding Concurrency to the Server Map

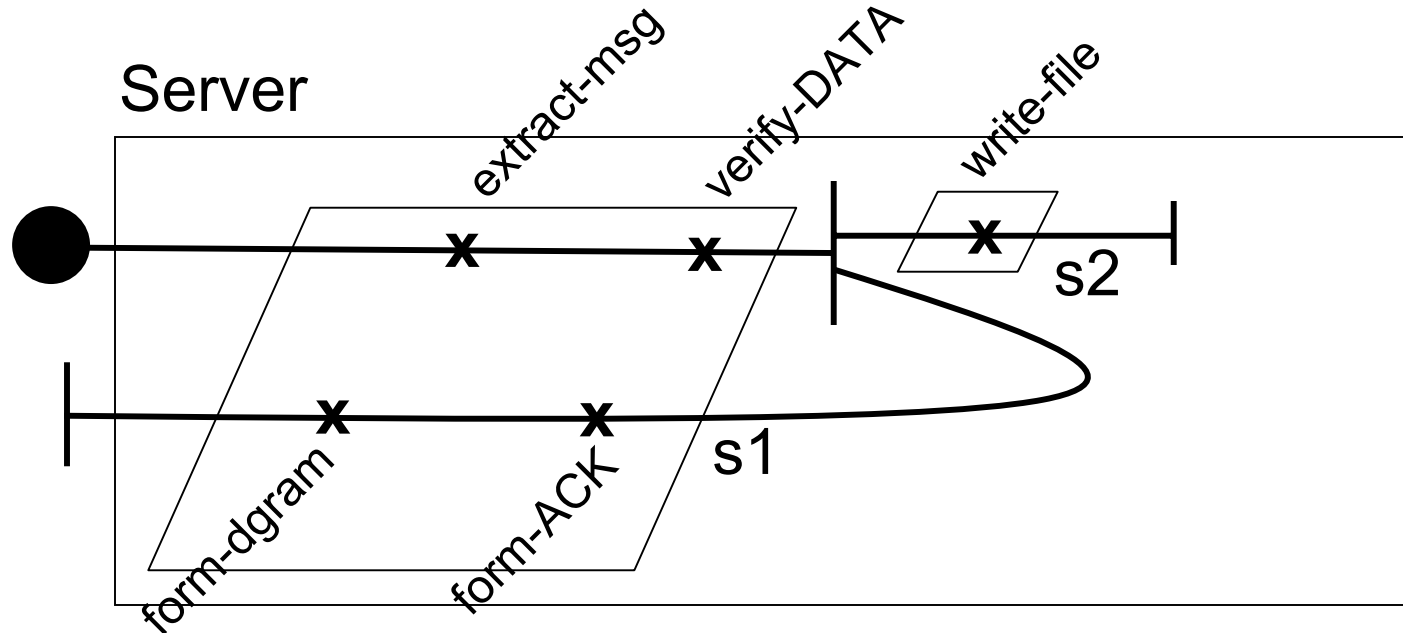
- The paths are concatenated using AND-fork notation:



AND-fork



Candidate Thread Architecture



- Allocate one thread to handle `write-file`, another thread to handle all other server responsibilities

Candidate Thread Architecture

- If TFTP DATA packets are placed in a buffer (not shown on the UCM) after `verify-DATA` occurs, the rate at which the `s1` thread can run through the **complete** `extract-msg/verify-DATA/form-ACK/form-dgram` path is unaffected by the rate at which `s2` write blocks of data to the file
 - in effect, we have a producer/consumer/bounded-buffer arrangement between the two threads
- Therefore, the addition of the `s2` thread, although it adds thread context switching overhead, could increase the overall server throughput

Exercise for the Student

- Build the UCMs that depict establishing a RRQ connection and steady-state file transfer from a server to the client
- Use the paths to identify the problem concurrency, then design a thread architecture that best implements that concurrency
- Build a *composite UCM* that combines the UCMs for reading and writing files into one map
 - what are the advantages/disadvantages of doing this, compared to drawing four separate UCMs?

Handling Concurrent Connections

- The lock-step nature of TFTP means that a client does not attempt to maximize the traffic over the network
 - at any time, there is at most one DATA packet or one ACK packet travelling between a client and the server
- If we remove the constraint that the server handle only one connection at a time; the time that a server spends waiting for DATA packets from one client could instead be used to handle connections with other clients

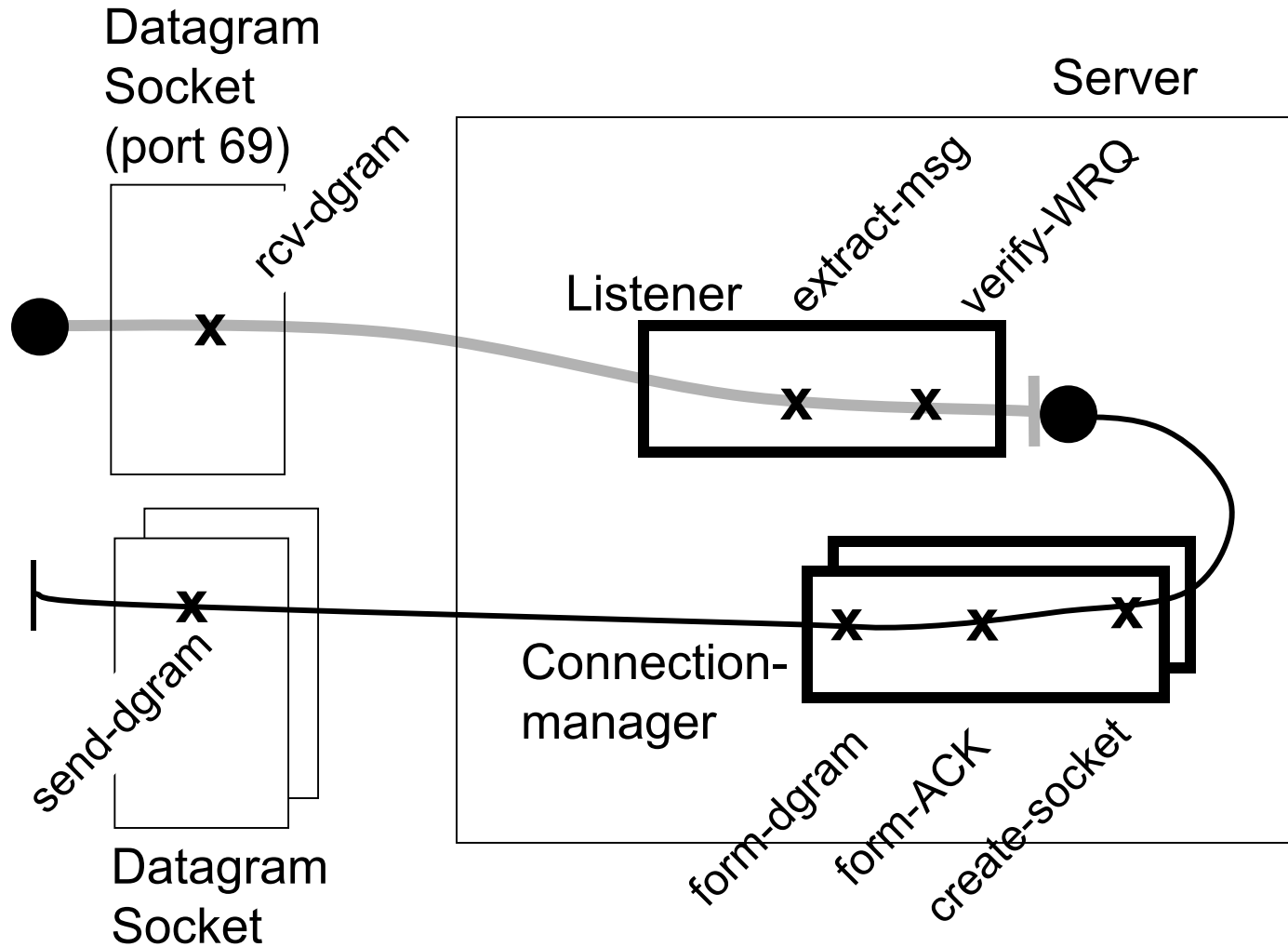
Handling Concurrent Connections

- It would be difficult for our 2-thread server to support concurrent connections with different clients
 - the server would have to poll port 69 for incoming requests and the other ports (one per connection) for incoming DATA packets
 - concurrency would be achieved by explicitly interleaving, in the thread's code, the execution of the responsibilities for the concurrent connections
 - all of this could be complex to program

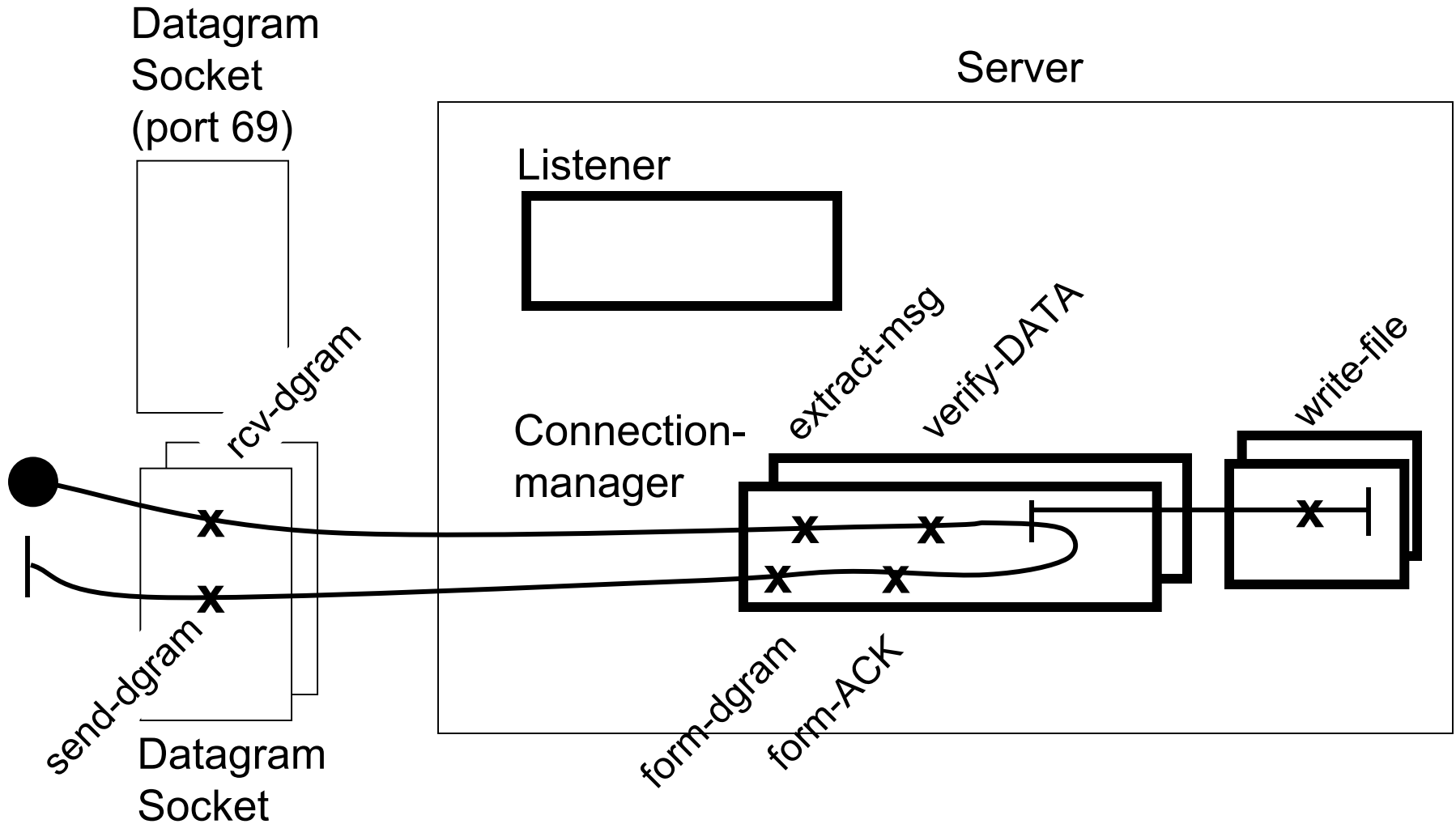
Handling Concurrent Connections

- We want to be able to have concurrent end-to-end traversals of the steady-state file transfer path (one traversal for each received TFTP DATA packet for each connection)
- We also want connection establishment to be concurrent with steady-state file transfer (so new connections can be established while others are in progress)
- We can accomplish this with a *multithreaded server* architecture

Multithreaded Server Architecture



Multithreaded Server Architecture



Multithreaded Server Architecture

- On the UCM shown on the previous slide, we need to document that the *listener* thread (handling the thick gray path), after verifying that a valid WRQ packet has been received, creates a new *connection-manager* thread to handle the create-socket, form-ACK, and form-dgram responsibilities, and that this new thread will handle the server-side responsibilities for the subsequent file transfer
- In the meantime, the listener thread can wait for a request from another client

Exercise for the Student

- Generalize your solution for reading a file from a server to the client to support a multithreaded server; i.e., one that supports concurrent connections
 - rework the UCMs, then redesign the thread architecture to support the additional concurrency