
SYSC 3303 Real-Time Concurrent Systems

TFTP Client-Server Design Part 2

- Copyright © 2000-2003 D.L. Bailey, Copyright 2002-2005 © L.S. Marshall, Systems and Computer Engineering, Carleton University
- revised March 1st, 2005

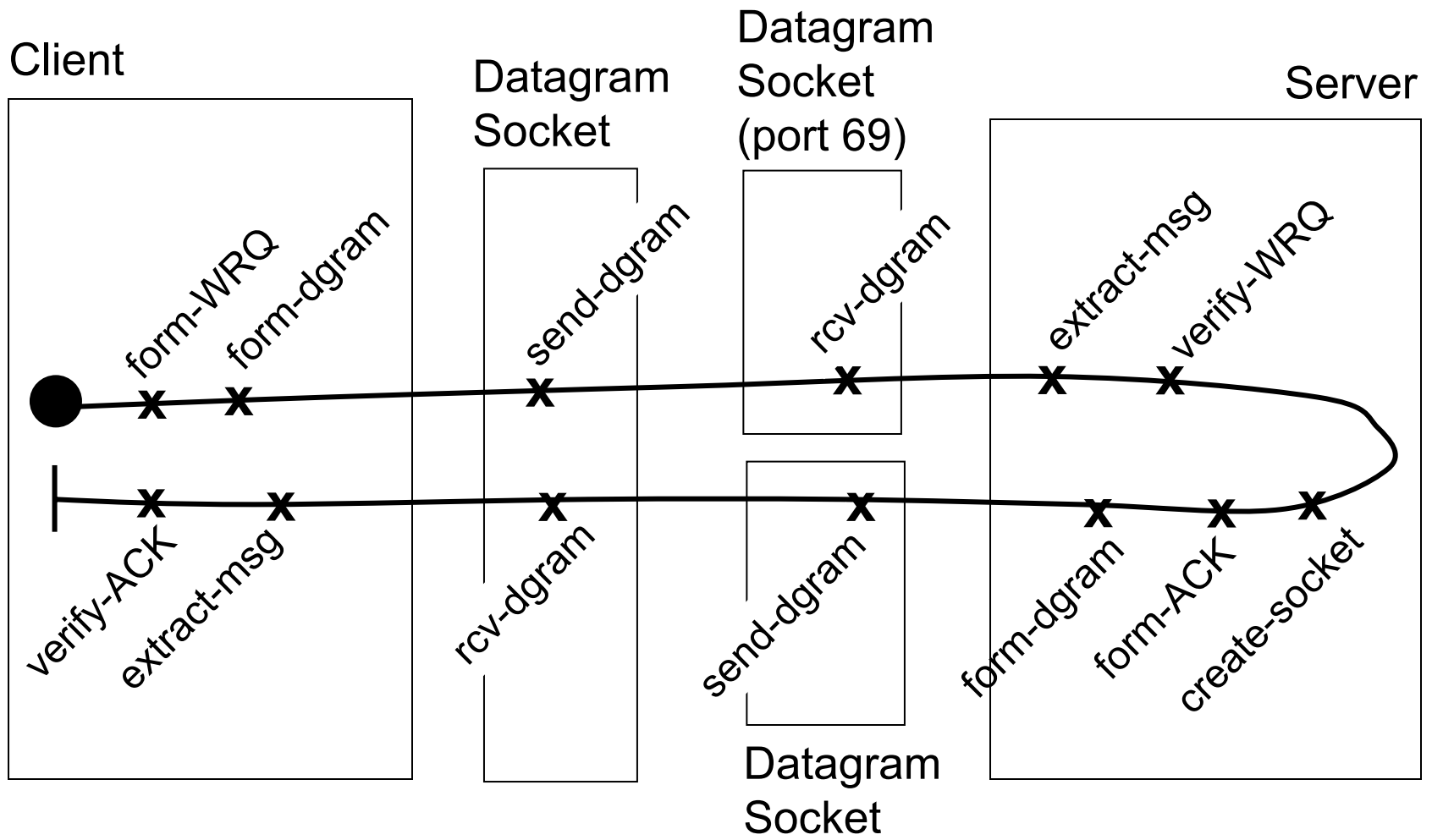
What's In This Set of Slides?

- We'll show how to use UCMs to model the cause-effect scenarios that result in ERROR packets being sent while writing a file from a client to a server
- We'll show how to use UCMs to model the wait for acknowledgment/timeout/retransmit protocol used by TFTP to recover when packets are lost
- We'll show how to use UCMs to reason about the behaviour of a TFTP client & server when packets are delayed or duplicated
- We'll show how to use UCMs to visualize the Sorcerer's Apprentice Bug & its solution

ERROR Packets

- Recall that errors detected during TFTP file transfer result in an ERROR packet being sent by the host that detects the error to the host at the other end of the connection
- The next several slides use UCMs to model the cause-effect paths through the system when errors occur while a client writes a file to a server

UCM: Error-free WRQ



WRQ Errors

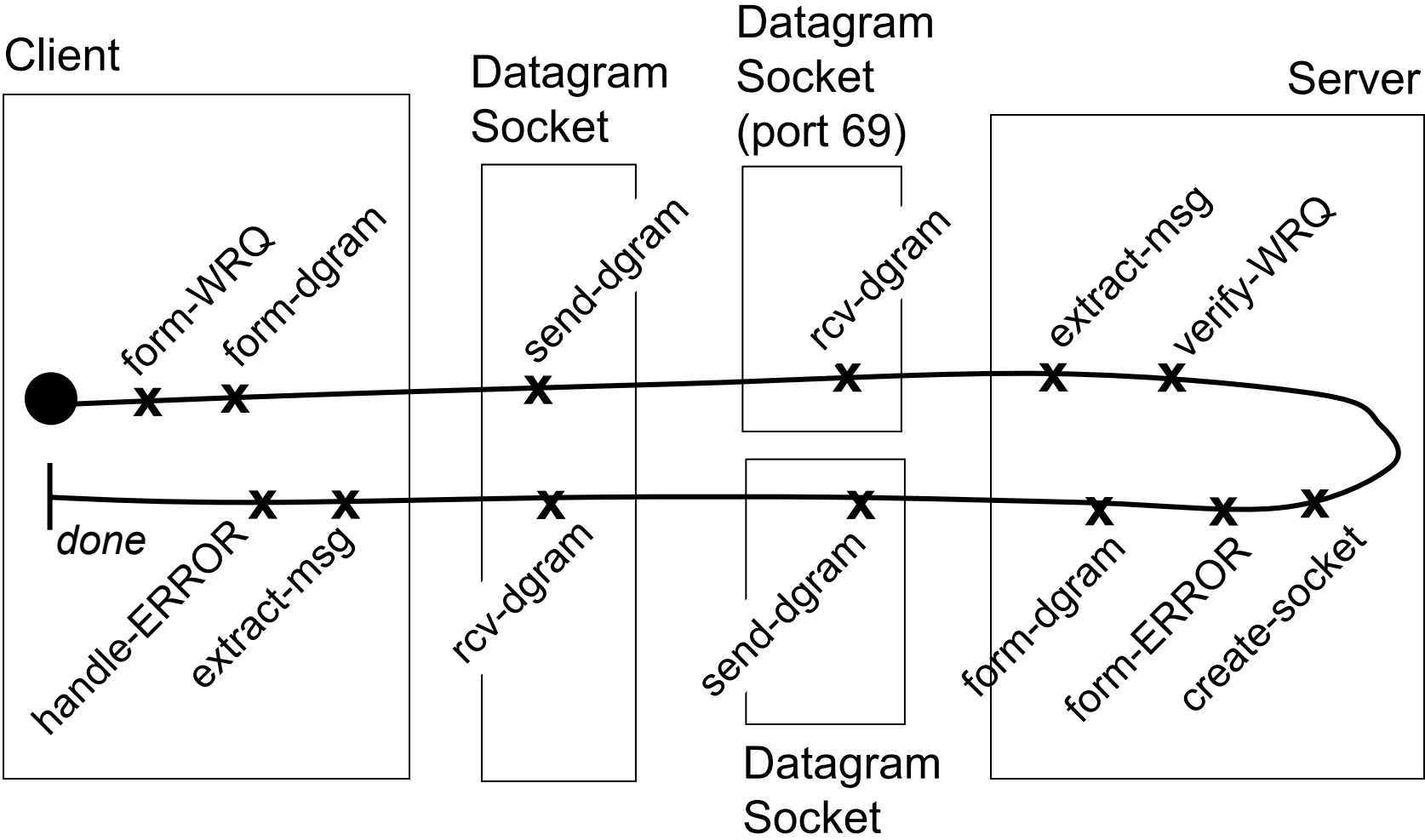
- Cause: the server receives a message through port 69 that is not a properly formed WRQ packet (invalid opcode, filename, or mode fields)
- Effect: the server creates an ERROR packet with error code 4 (illegal TFTP operation), sends it to the client, and terminates the connection

WRQ Errors

- Cause: the server receives a WRQ packet, but cannot open the specified file for writing
- Effect: the server creates an ERROR packet with error code 2 (access violation), error code 3 (disk full), or error code 6 (file already exists - do we really need this one?); sends it to the client; and terminates the connection
 - errors detected depend on the file I/O library provided by the programming technology

UCM for WRQ Error Handling

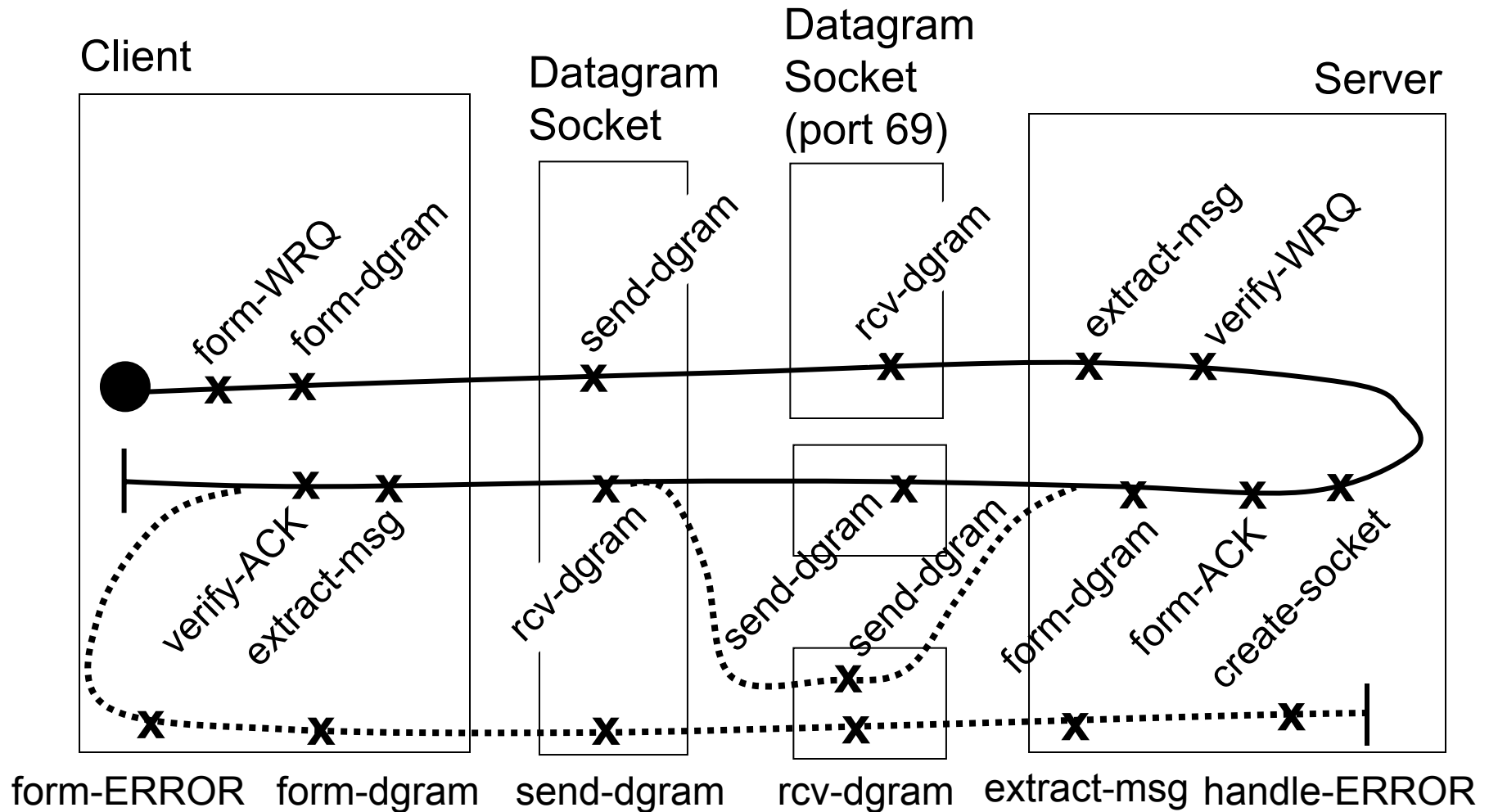
- New responsibilities (see next slide):
 - form-ERROR: prepare a TFTP ERROR packet
 - handle-ERROR: determine that the received message is a TFTP ERROR packet and handle the error condition (release resources, terminate connection)



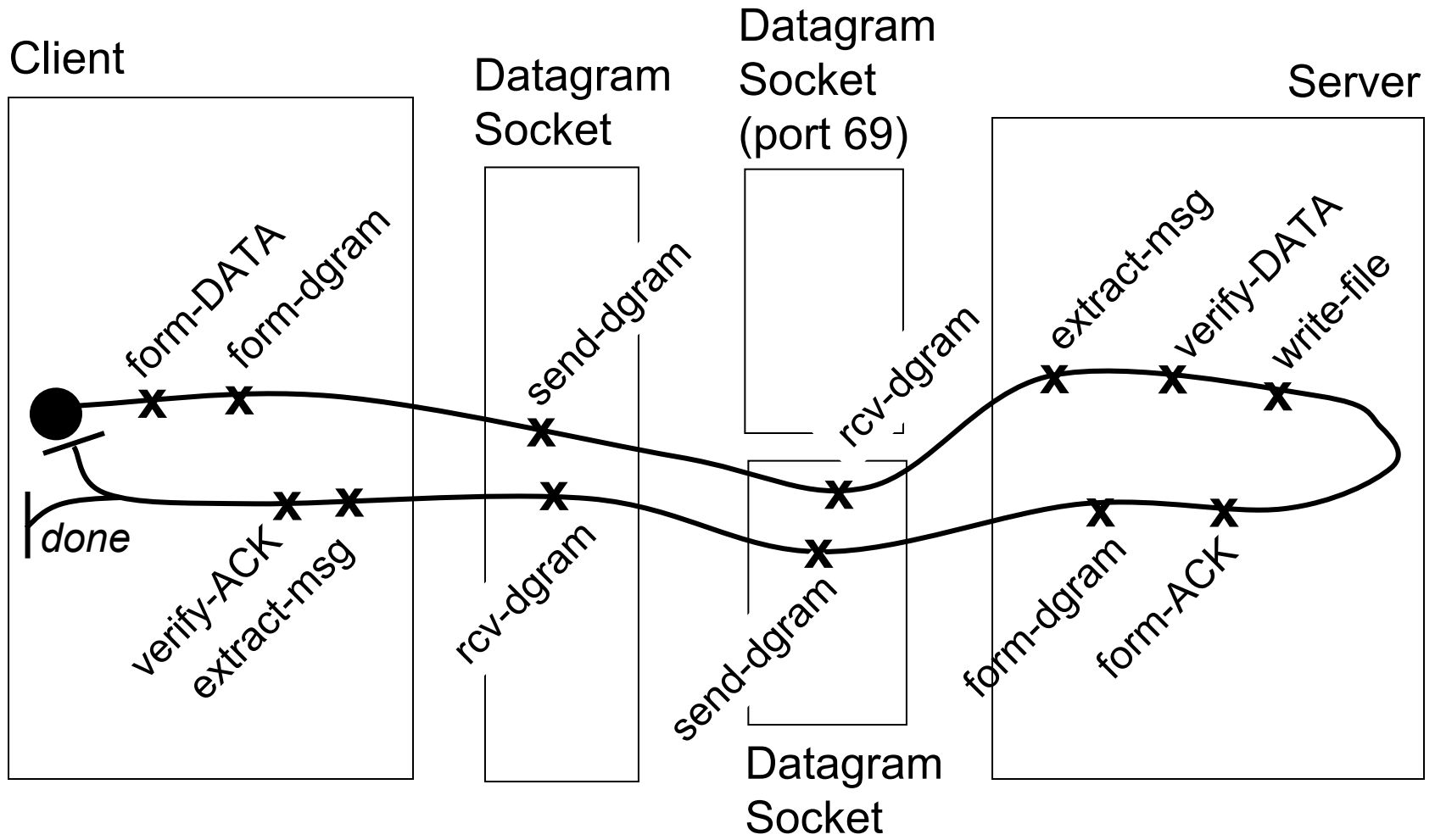
WRQ Errors (2)

- Cause: a WRQ packet is duplicated between the client and server. The server establishes two connections, and sends two ACK packets (with different TIDs) to the client
- Effect: the client rejects the second ACK packet by preparing an ERROR packet with error code 5 (unknown transfer ID) and sends it to the server via the TID specified in the second ACK packet

UCM: WRQ Error Handling (2)



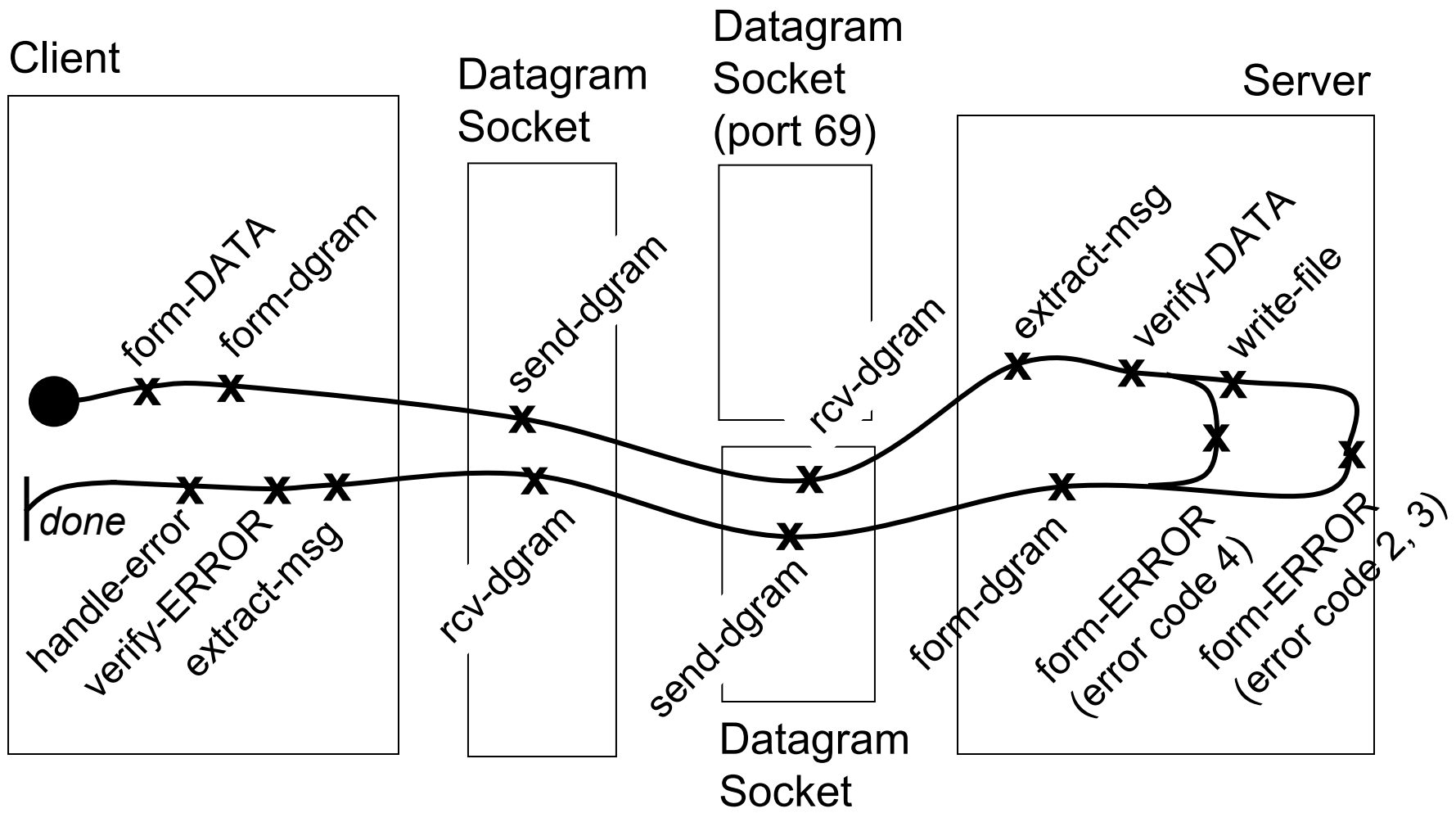
UCM: Error-free Write Connection



Write Connection Errors

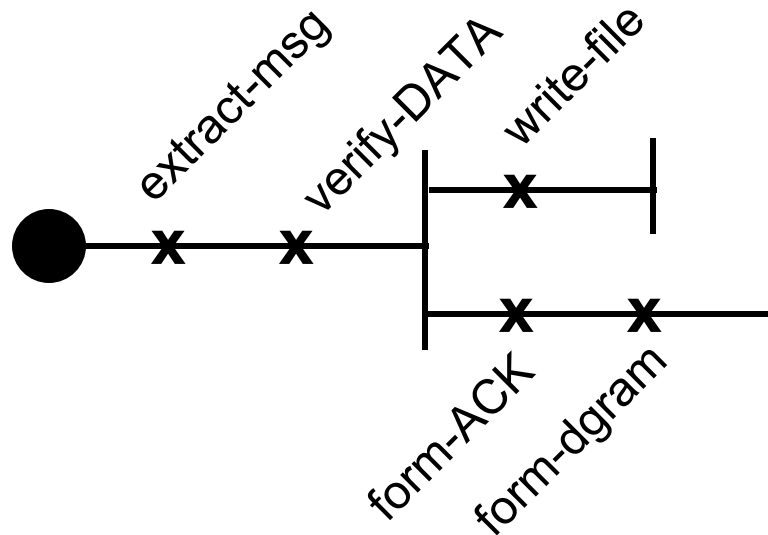
- Cause: during the file transfer,
 - the server receives a TFTP packet that is not a DATA or ERROR packet (error 4)
 - the server receives a DATA packet with an out-of-sequence block number (error 4?)
 - the disk becomes full (error 3)
 - the server loses access rights to the file (error 2)
- Effect: the server creates an ERROR packet with the appropriate error code, sends it to the client, and terminates the connection

UCM: Write Connection Error Handling



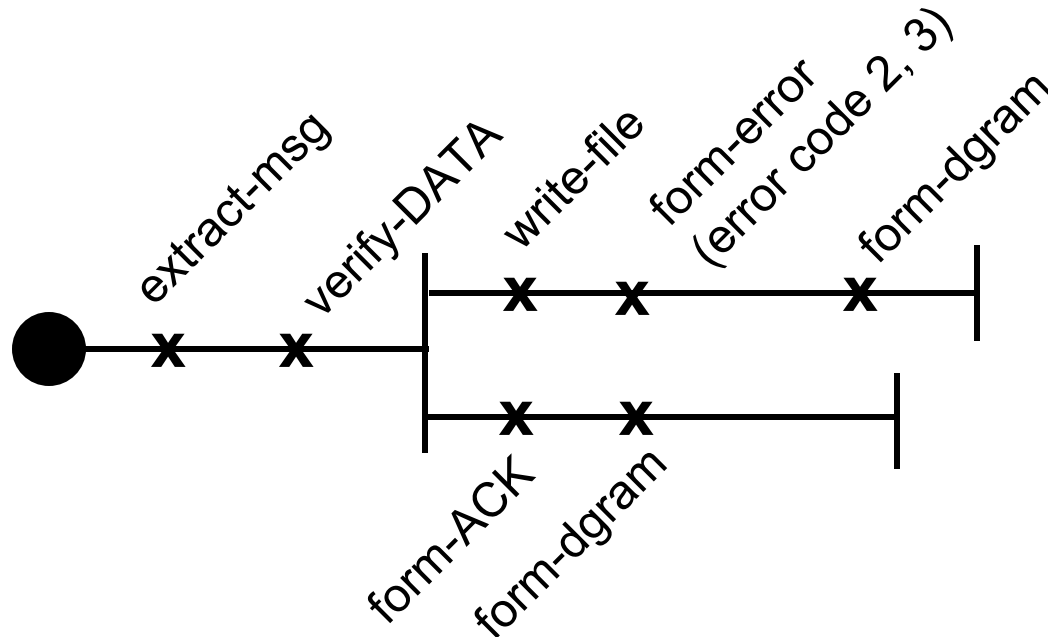
Additional Concurrency in the Server Map

- Recall that we considered making the write-file responsibility concurrent with preparation and sending of the ACK packet



Concurrency Implications

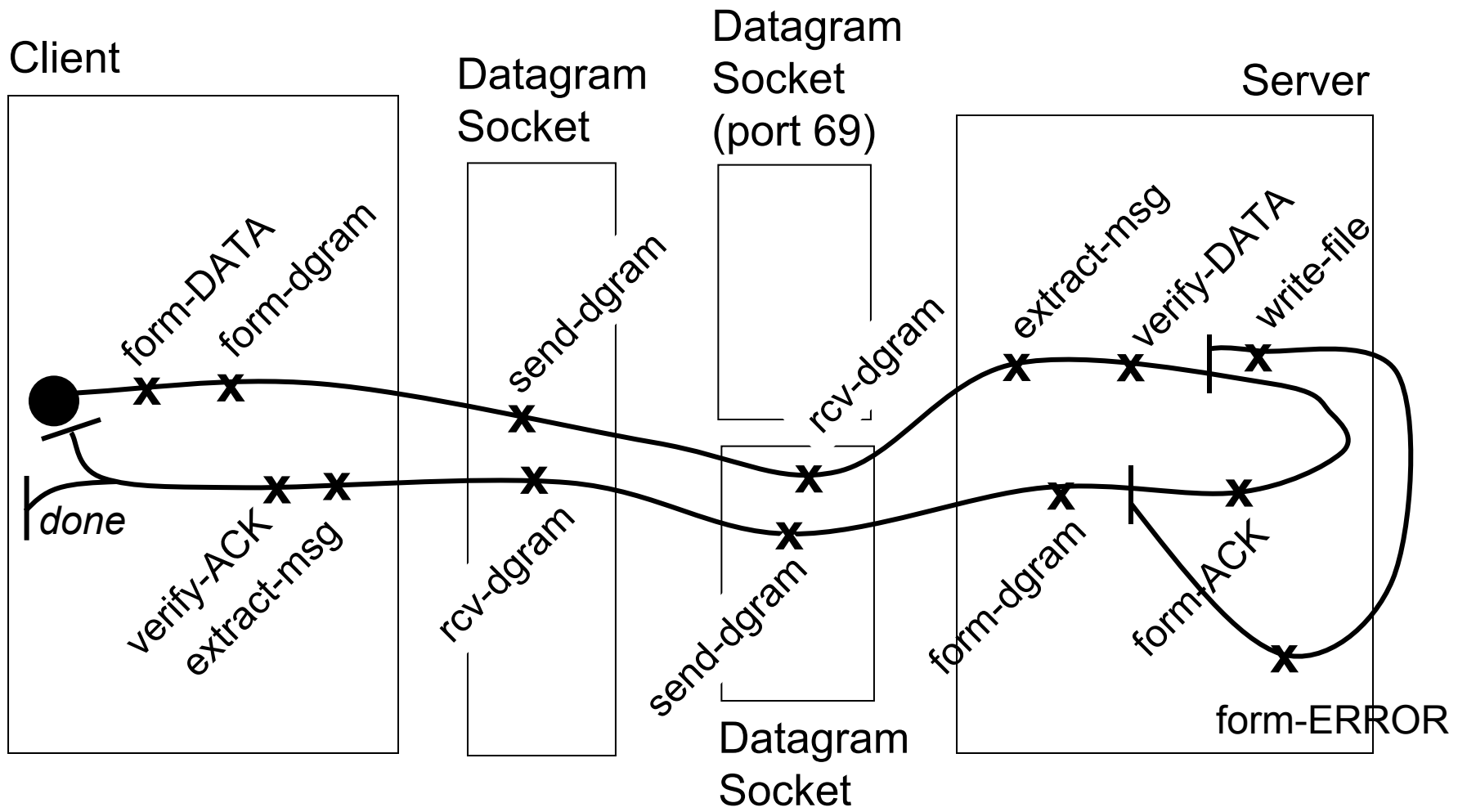
- Now that we're considering ERROR packets, is this still feasible?



Concurrency Implications

- An error while writing the file will result in an ACK packet and an ERROR packet being sent for the same DATA packet
- If data blocks are buffered before being written, the ERROR packet associated with a particular DATA packet may be prepared and sent well after the ACKs for subsequent DATA packets have been sent
- When will this not be a problem?
- When will this be a problem?

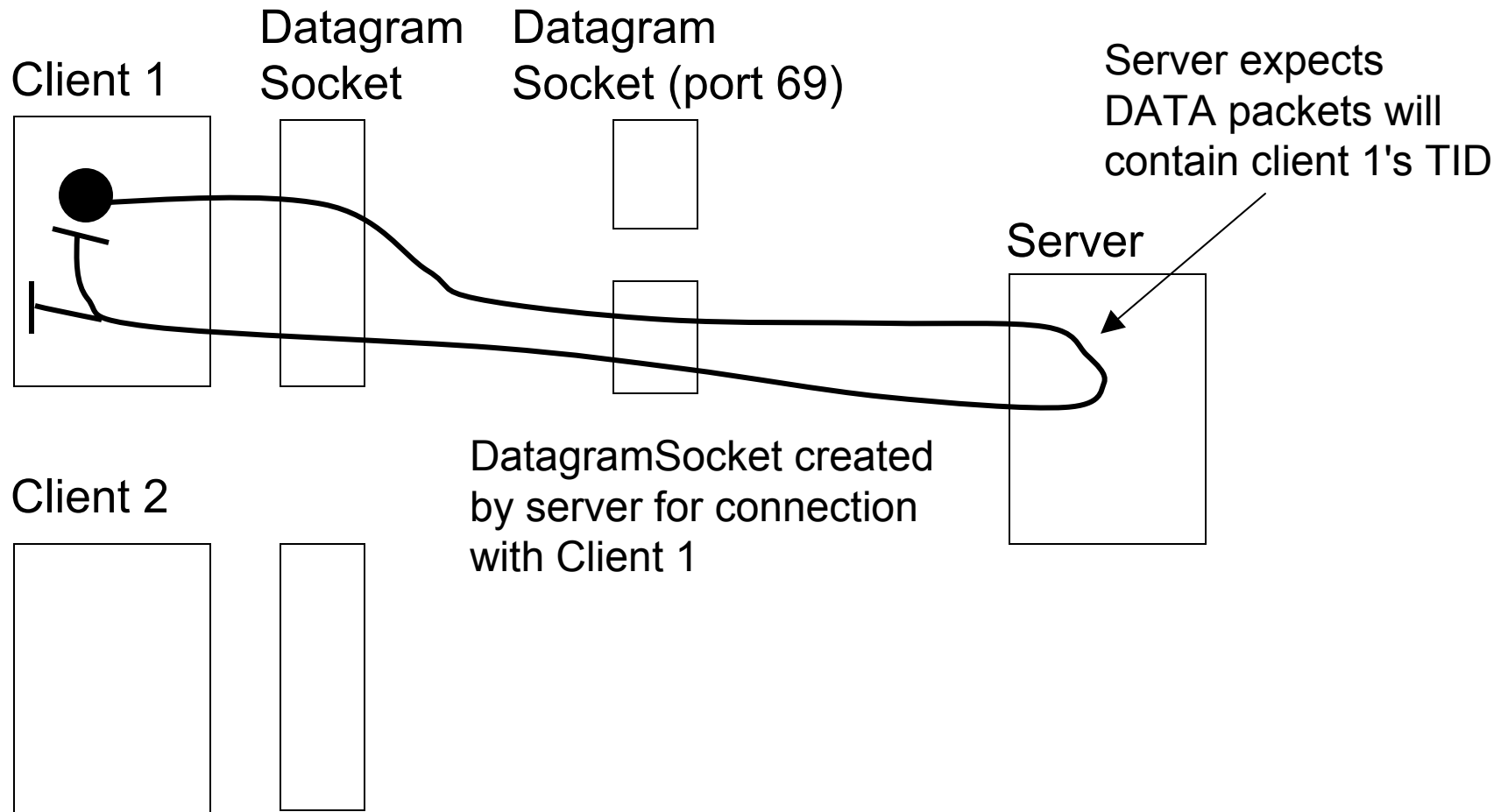
Concurrency Implications



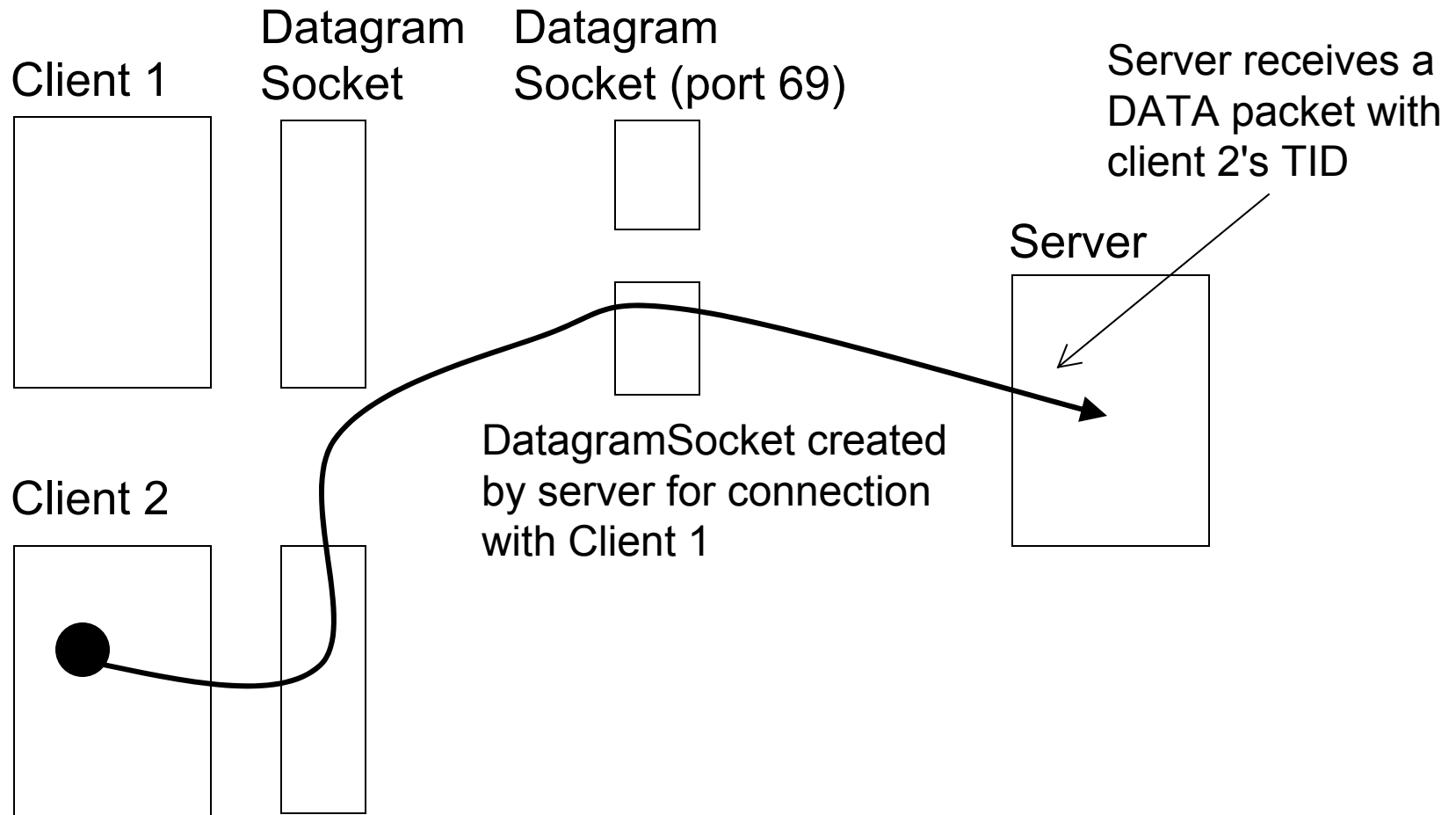
Write Connection Errors (2)

- Cause: the source TID of a DATA packet received by the server is incorrect

Expected Scenario



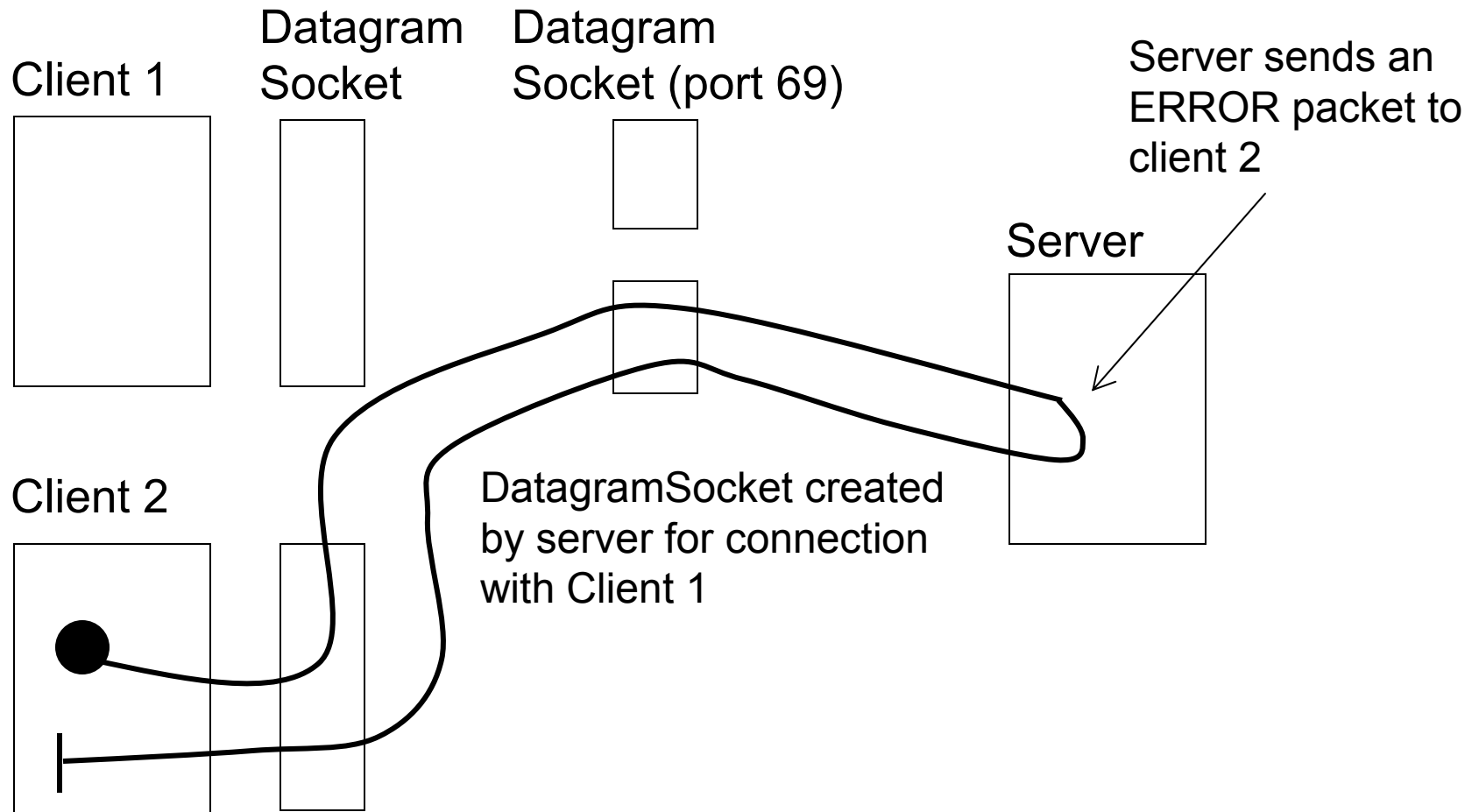
Unexpected Scenario



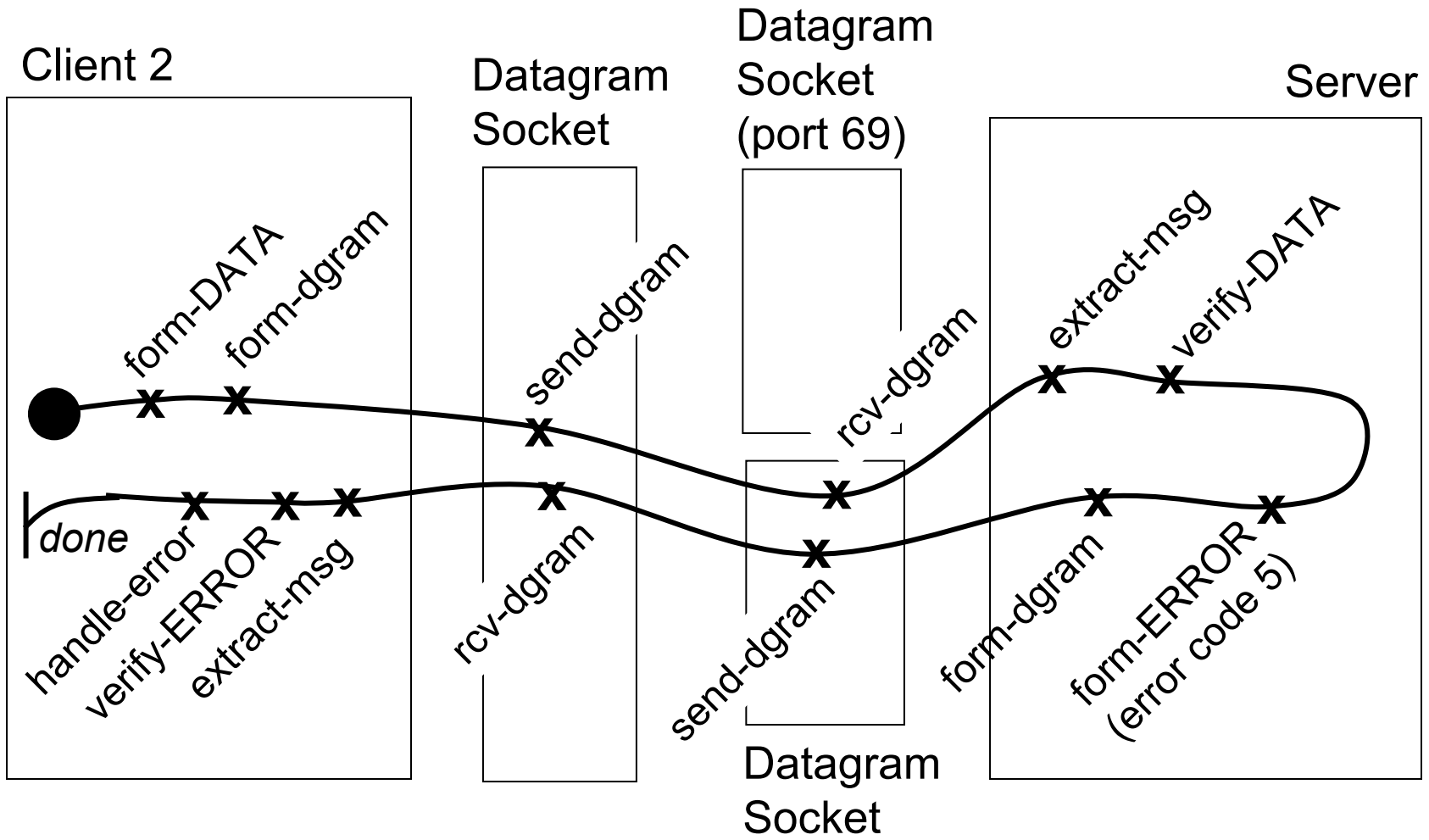
Write Connection Errors (2), cont'd

- Effect: the server creates an ERROR packet with error code 5 (unknown transfer ID) and sends it to the originating host (in this case, client 2), but does not terminate the connection with client 1

Error Handling



UCM: Write Connection Error Handling (2)



Write Connection Errors (3)

- Cause: the source TID of an ACK packet received by the client is incorrect
 - this is similar to error scenario just presented, so detailed UCMs will not be presented here
- Effect: the client creates an ERROR packet with error code 5 (unknown transfer ID) and sends it to the originating host, but does not terminate the connection to the server which is the destination of the file transfer

Event Handling

- A real-time system must be able to deal with these 3 situations:
 - expected events arrive
 - unexpected events arrive
 - expected events fail to arrive

TFTP Event Handling

- The major events in TFTP are the arrival of TFTP packets encapsulated in UDP datagrams
- The client and server distinguish between expected and unexpected events by checking the TFTP packet type
- The client and server know when expected events fail to arrive by using the timeout facility provided by DatagramSocket

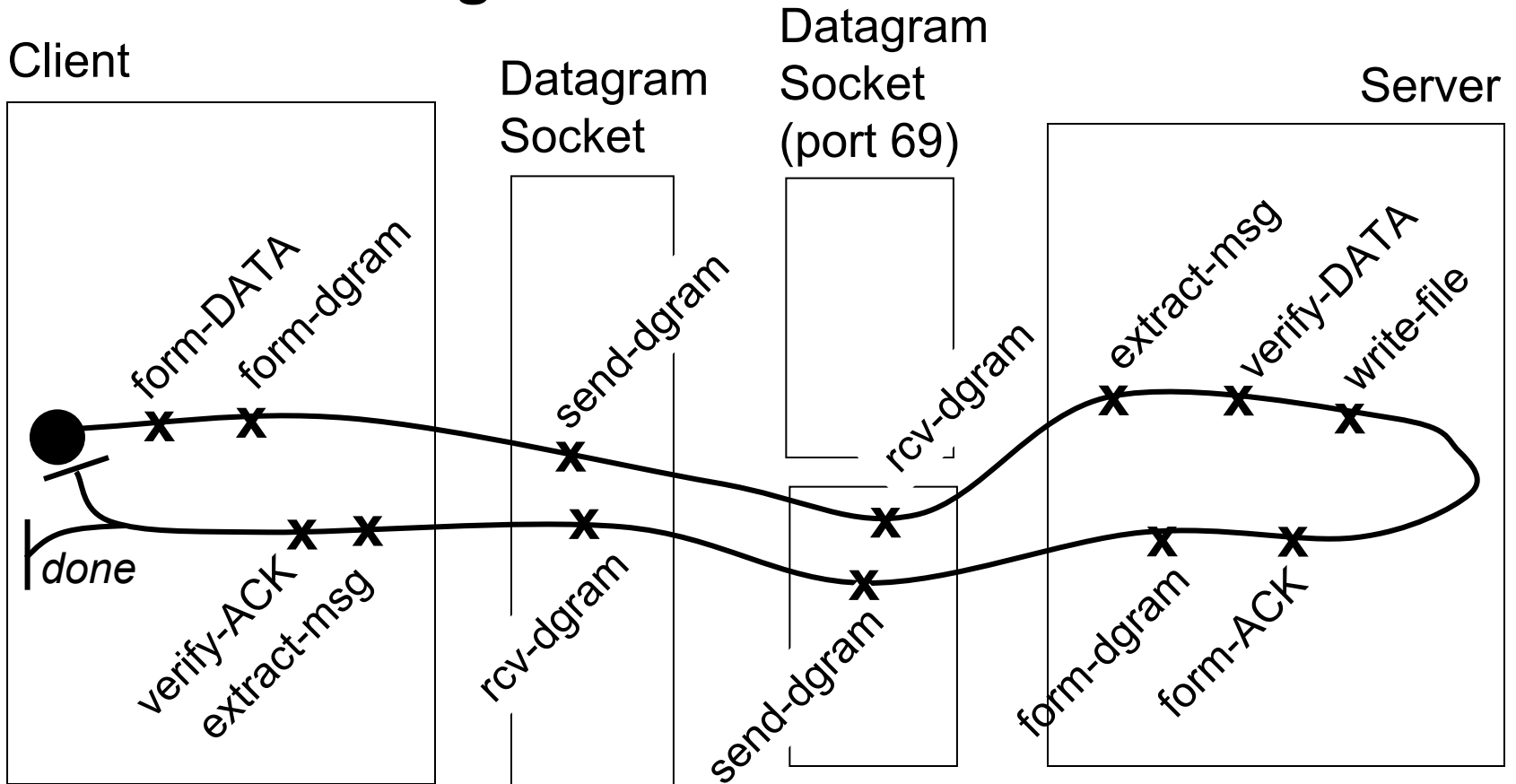
setSoTimeout()

- "Enable/disable `SO_TIMEOUT` with the specified timeout, in milliseconds. With this option set to a non-zero timeout, a call to `receive()` for this `DatagramSocket` will block for only this amount of time. If the timeout expires, a **`java.net.SocketTimeoutException`** is raised, though the `DatagramSocket` is still valid. The option **must** be enabled prior to entering the blocking operation to have effect. The timeout must be > 0 . A timeout of zero is interpreted as an infinite timeout."

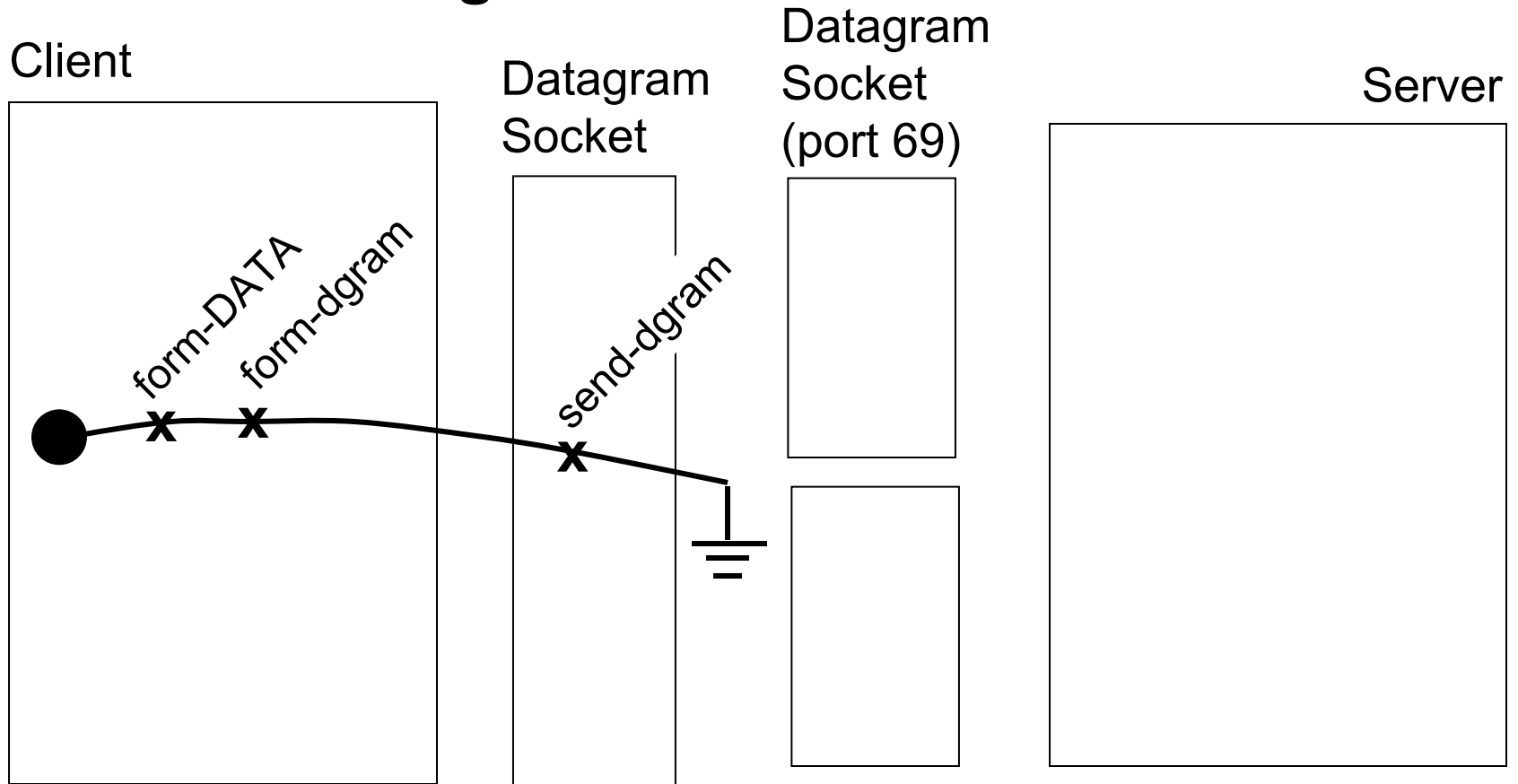
Dealing With Lost Packets

- We'll only consider steady-state file transfer from the client to the server (i.e., the precondition is that a WRQ connection has been established)
 - you should verify that the failure recovery mechanism will also handle packets lost when establishing a WRQ connection, a RRQ connection, and steady-state transfer from the server to the client (hint: draw the UCMs!)

Reasoning About Success and Failure

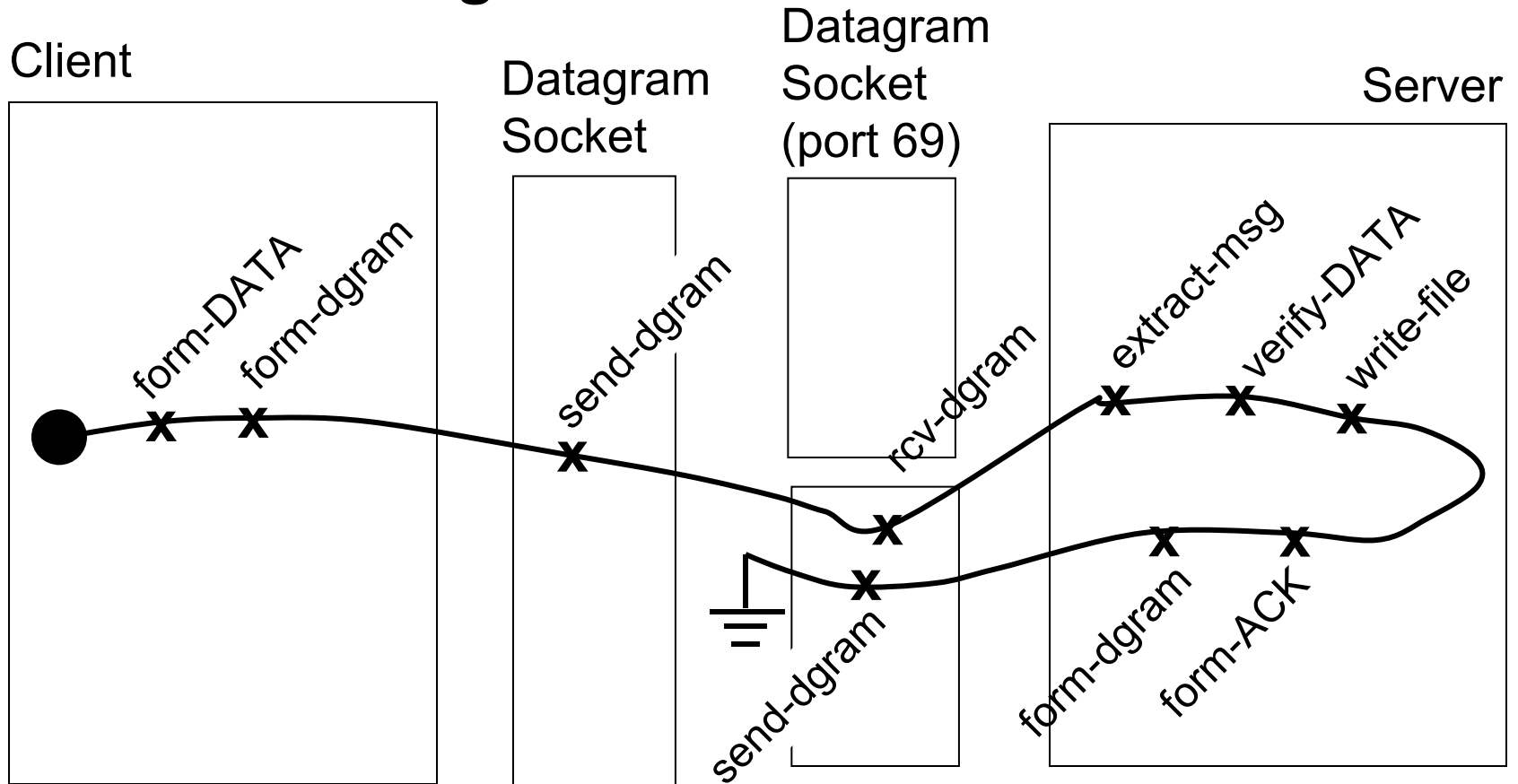


- Token trace for success scenario: send DATA packet, receive ACK



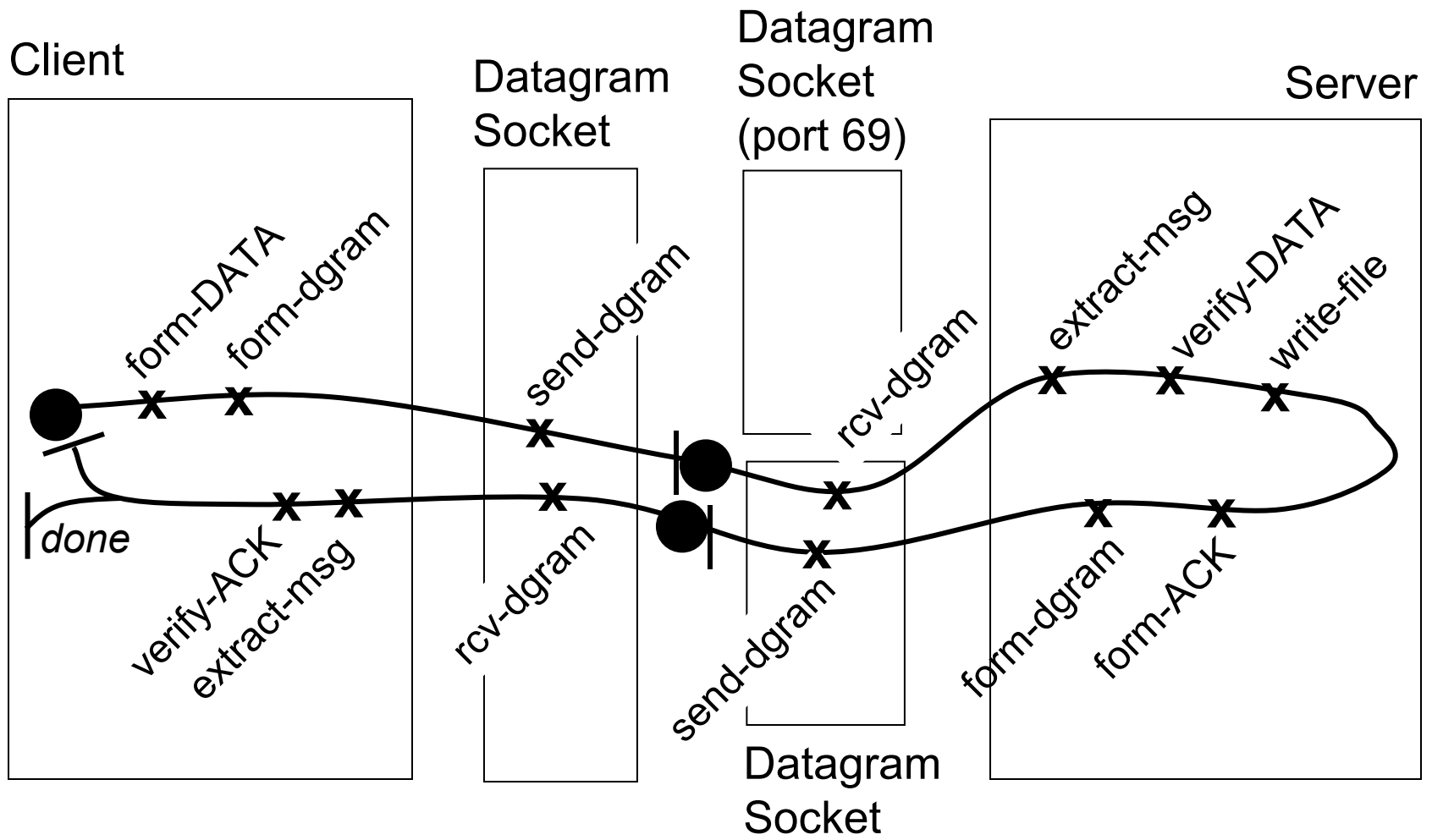
- Token trace for failure scenario: DATA packet lost

Reasoning About Success and Failure



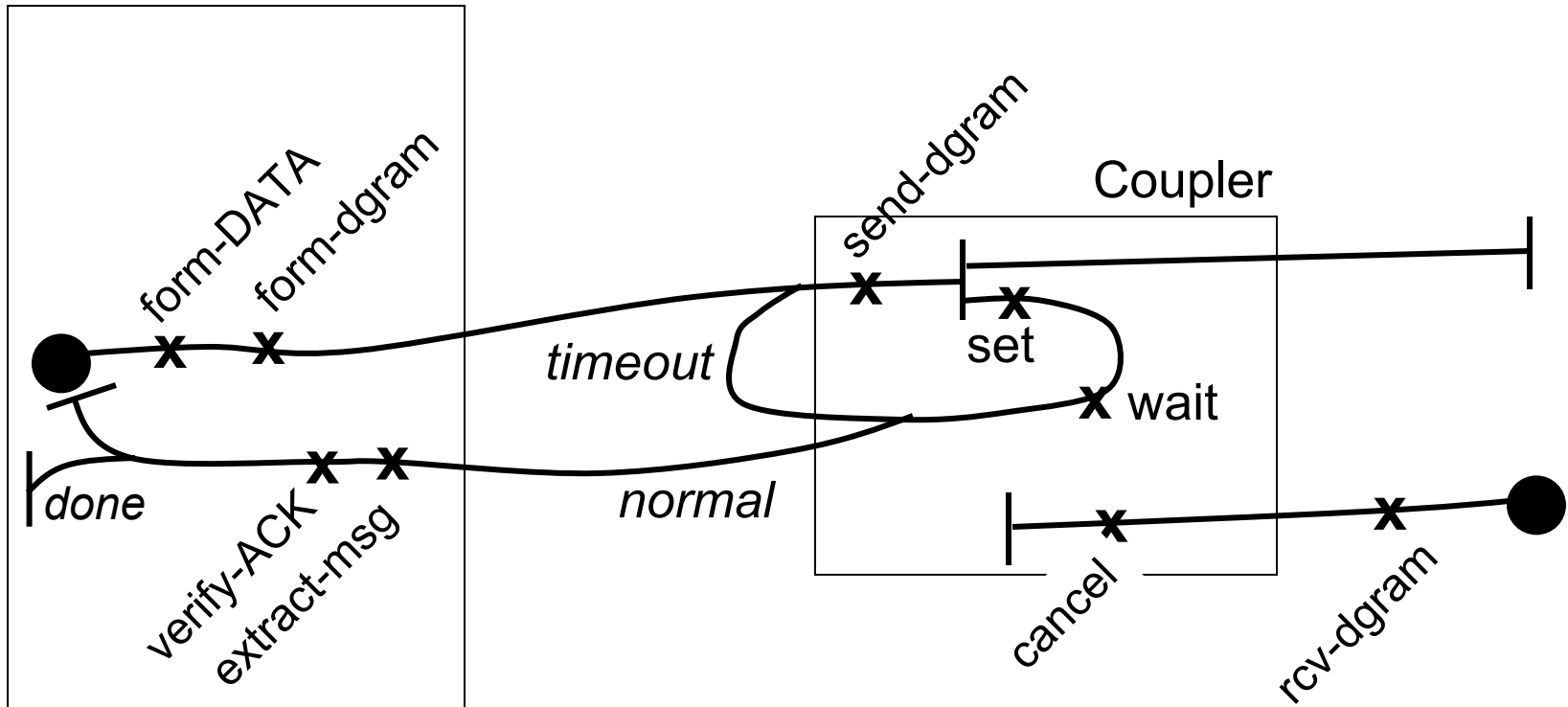
- Token trace for failure scenario: ACK packet lost

Recovery: Begin By Factoring the Map



Adding Failure Recovery to the Client Side

Client



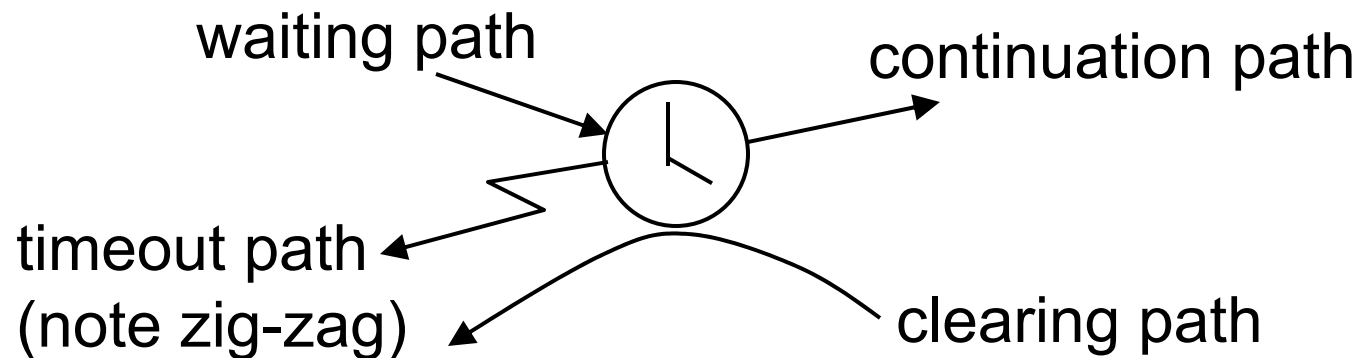
- If client times out while waiting for a packet, it retransmits the last DATA packet it sent

Coupler Component

- Coupler responsibilities
 - `set`: sets a timer
 - `wait`: wait for cancellation of waiting condition (by normal reception of ACK or timeout)
 - `cancel`: cancel waiting condition, stop timer
- *timeout* segment: leads to retransmission of DATA packet
- *normal* segment: packet received (path for ACK packet reception shown on the slide; path followed if an ERROR packet received not shown)

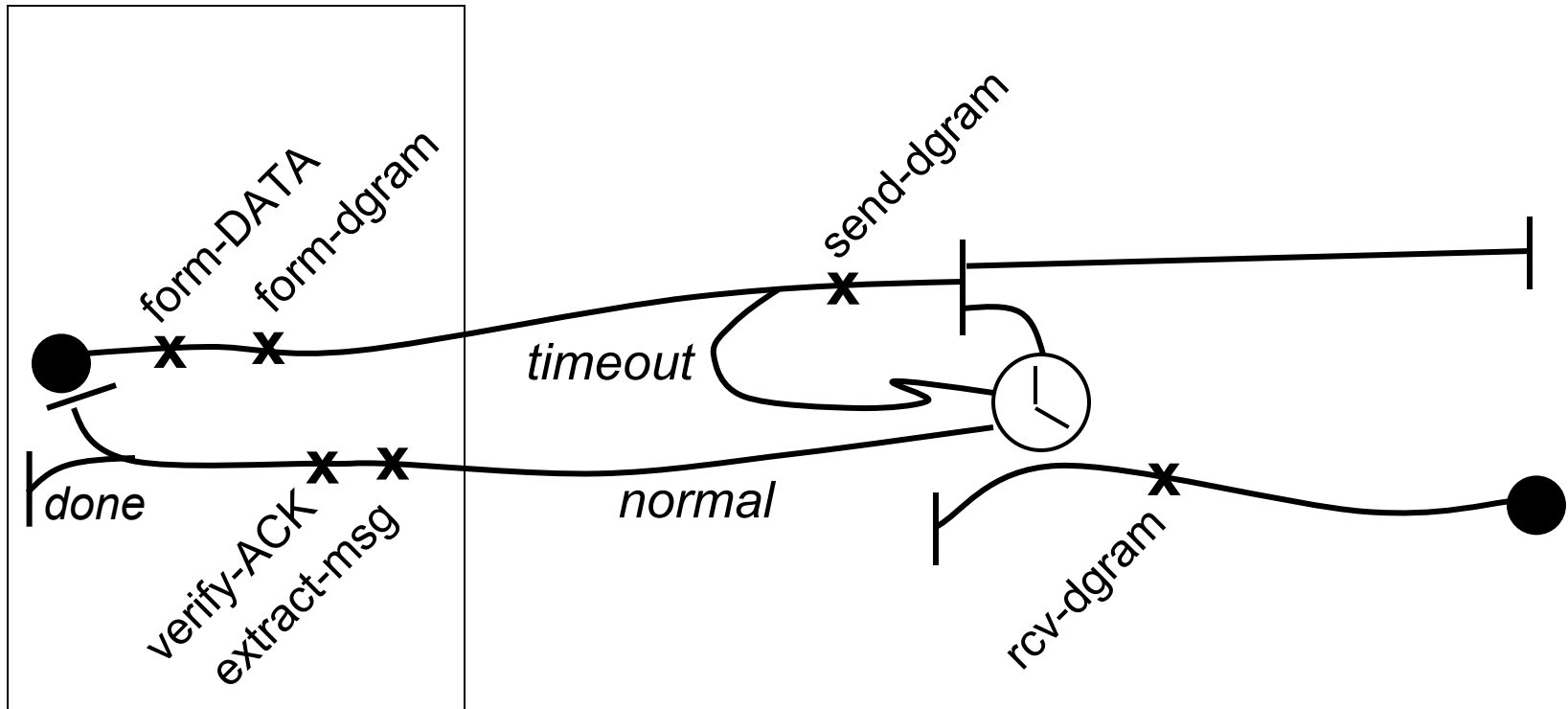
Visual Shorthand for Waiting With Timeout

- The coupler component shows asynchronous interpath coupling in a general way, but the timeout/recovery mechanism is sufficiently useful to warrant shorthand notation
- This notation indicates asynchronous interpath coupling with timeout (see UCM paper, Section 3.1)



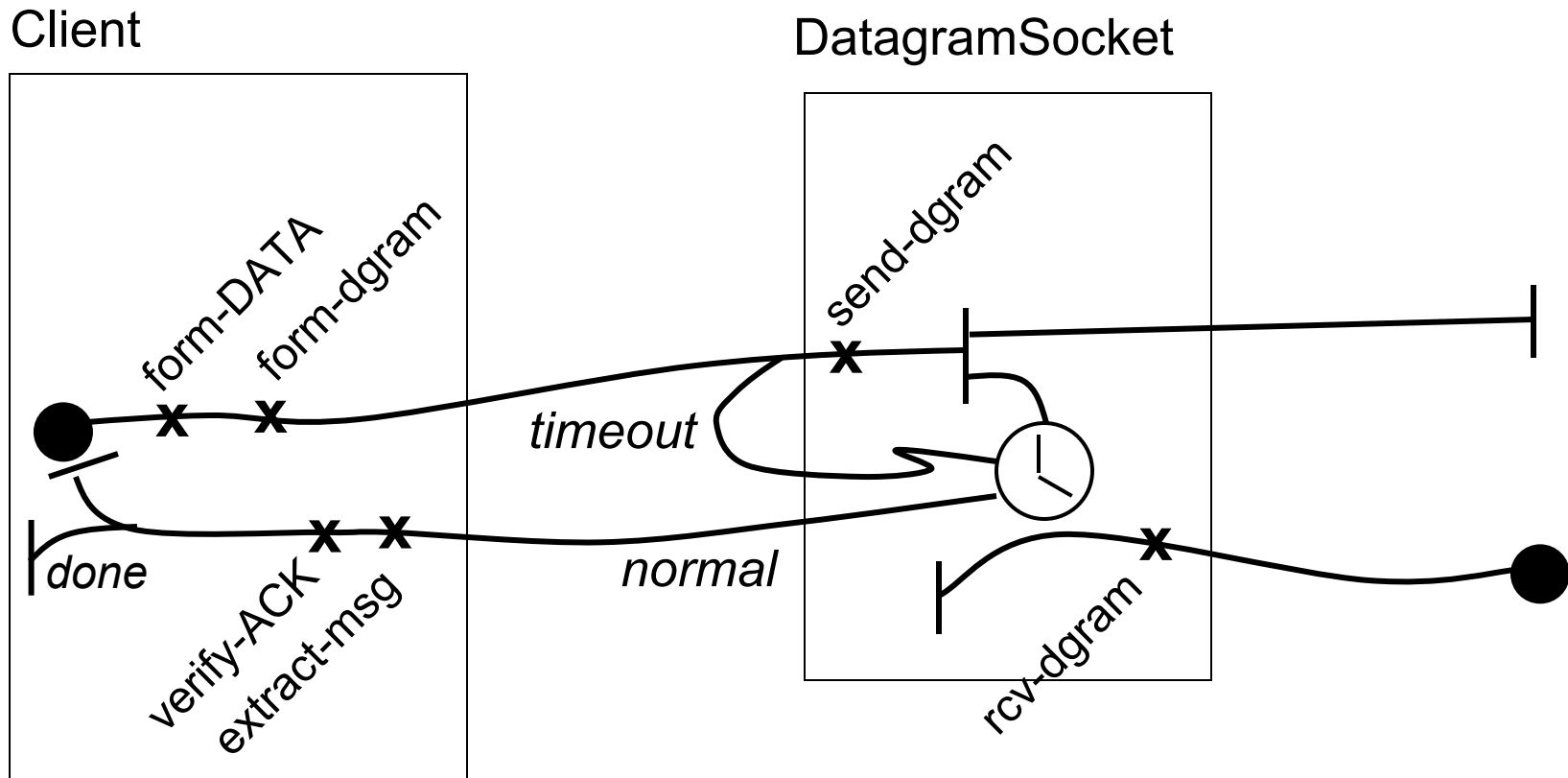
Failure Recovery Expressed With Visual Shorthand

Client



- If client times out while waiting for a packet, it retransmits the last DATA packet it sent

Timeouts and DatagramSockets

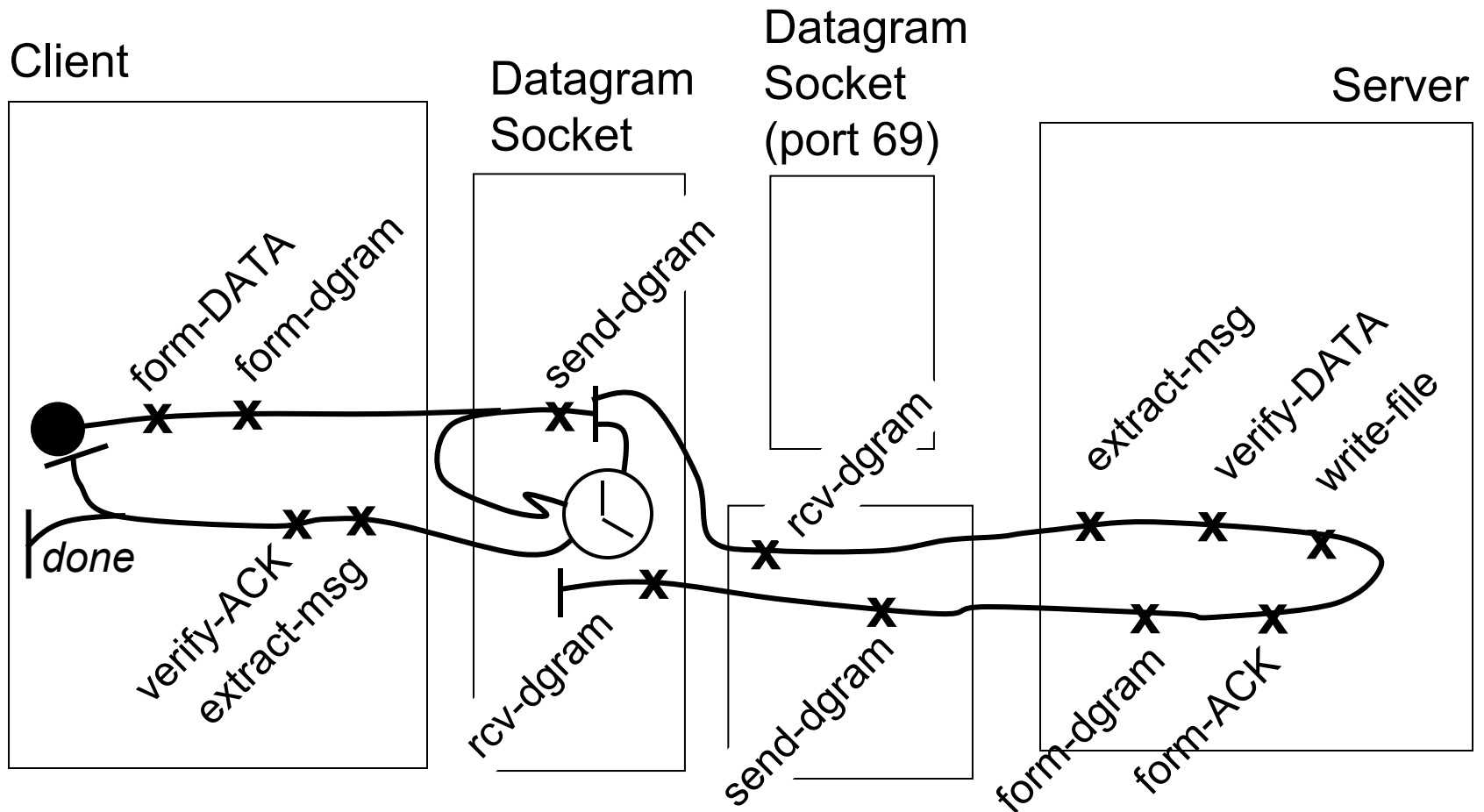


- Fortunately, wait with timeout capability is provided by Java's `DatagramSocket` class

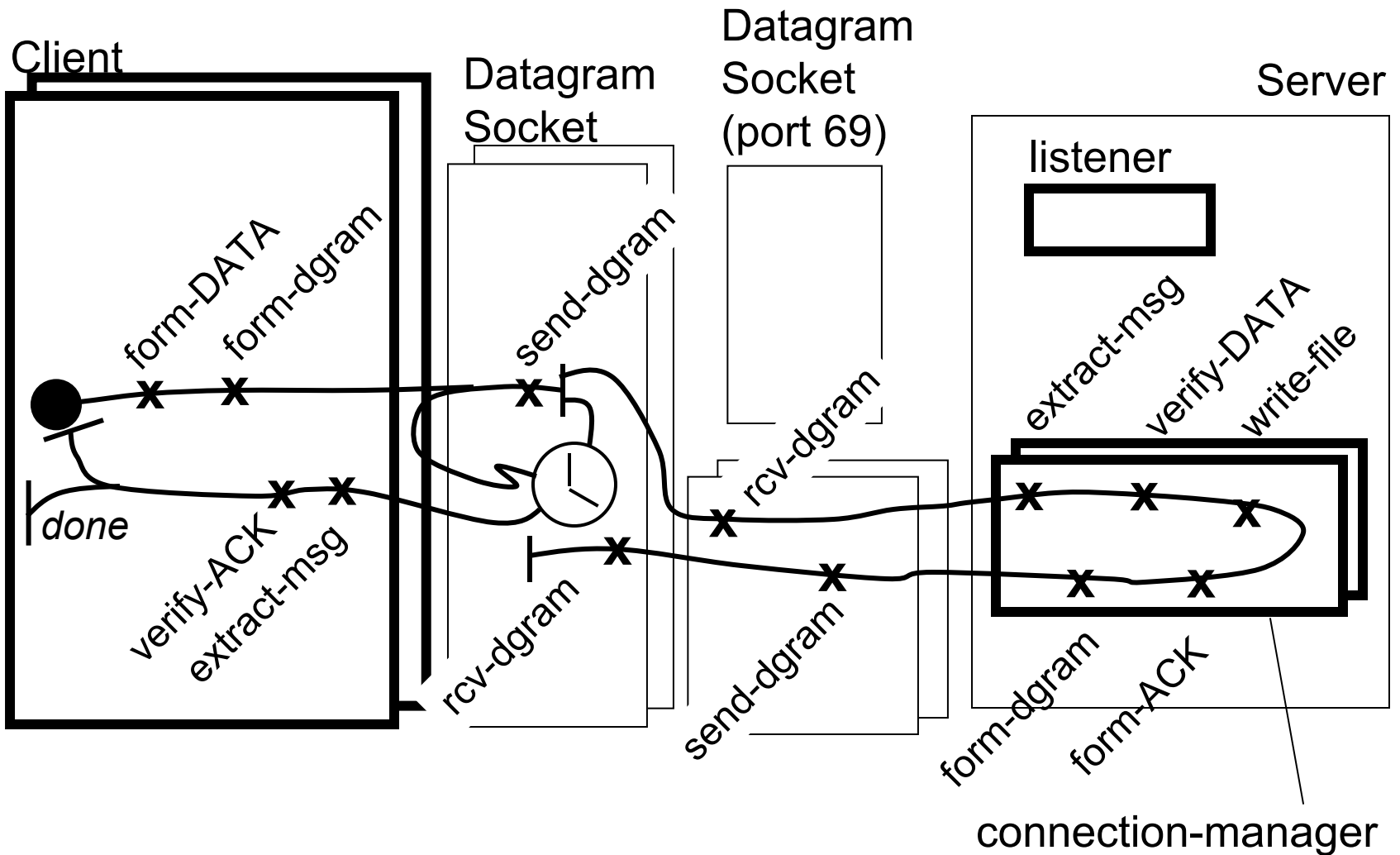
Timeout/Retransmit in the Server?

- Recall from our discussion of the Sorcerer's Apprentice Bug that we need a retransmission timer on the side that sends DATA packets (will timeout and retransmit if ACK not received), but we do not need a retransmission timer on the side that sends ACK packets (a retransmitted ACK is always ignored, so why retransmit it?)
- So, in the write connection UCM, we do not need to show timeout/retransmit on the server side
 - we will need to show this on the read connection UCM

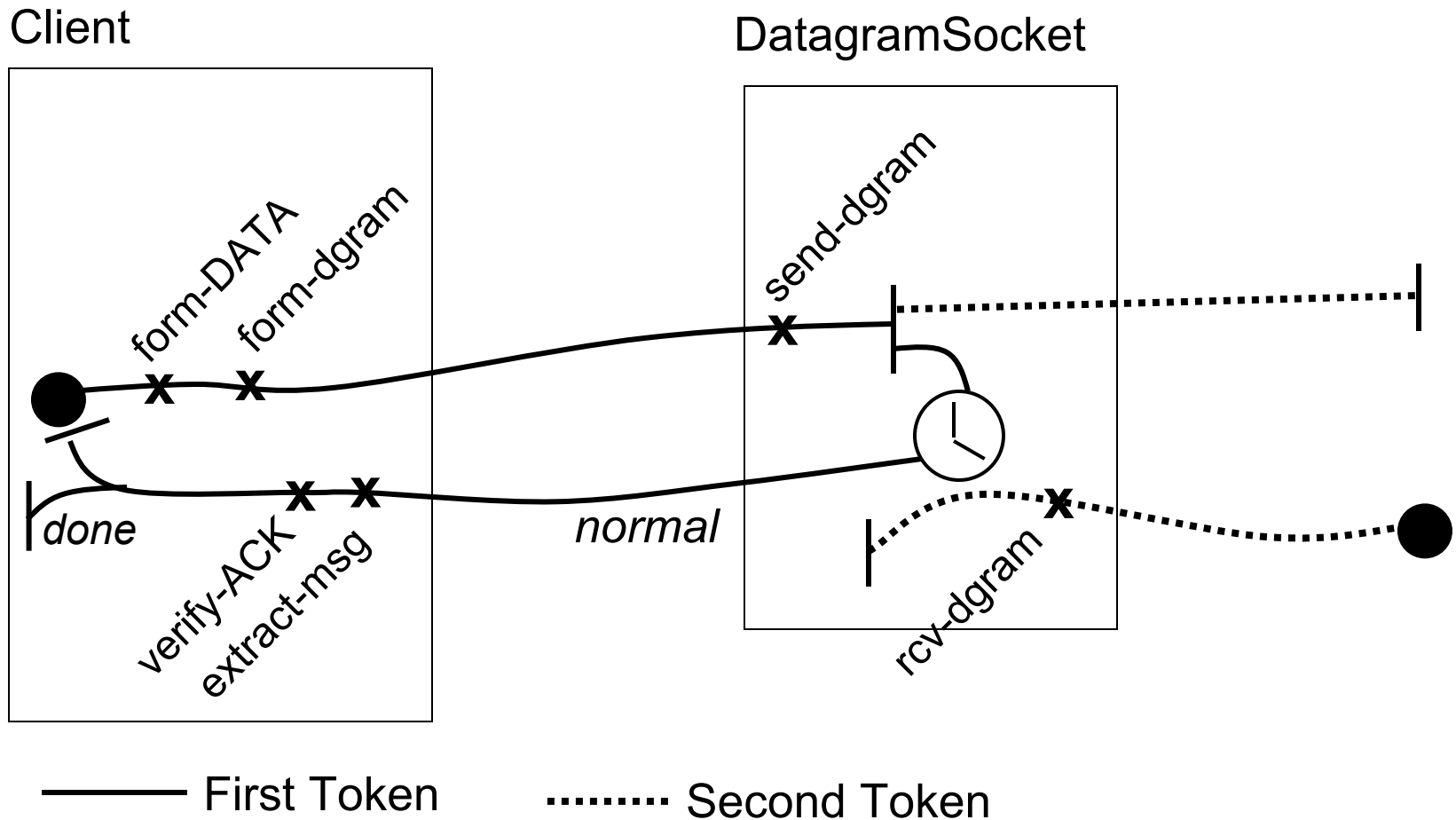
Concatenate the Client and Server Sides



Multithreaded Server



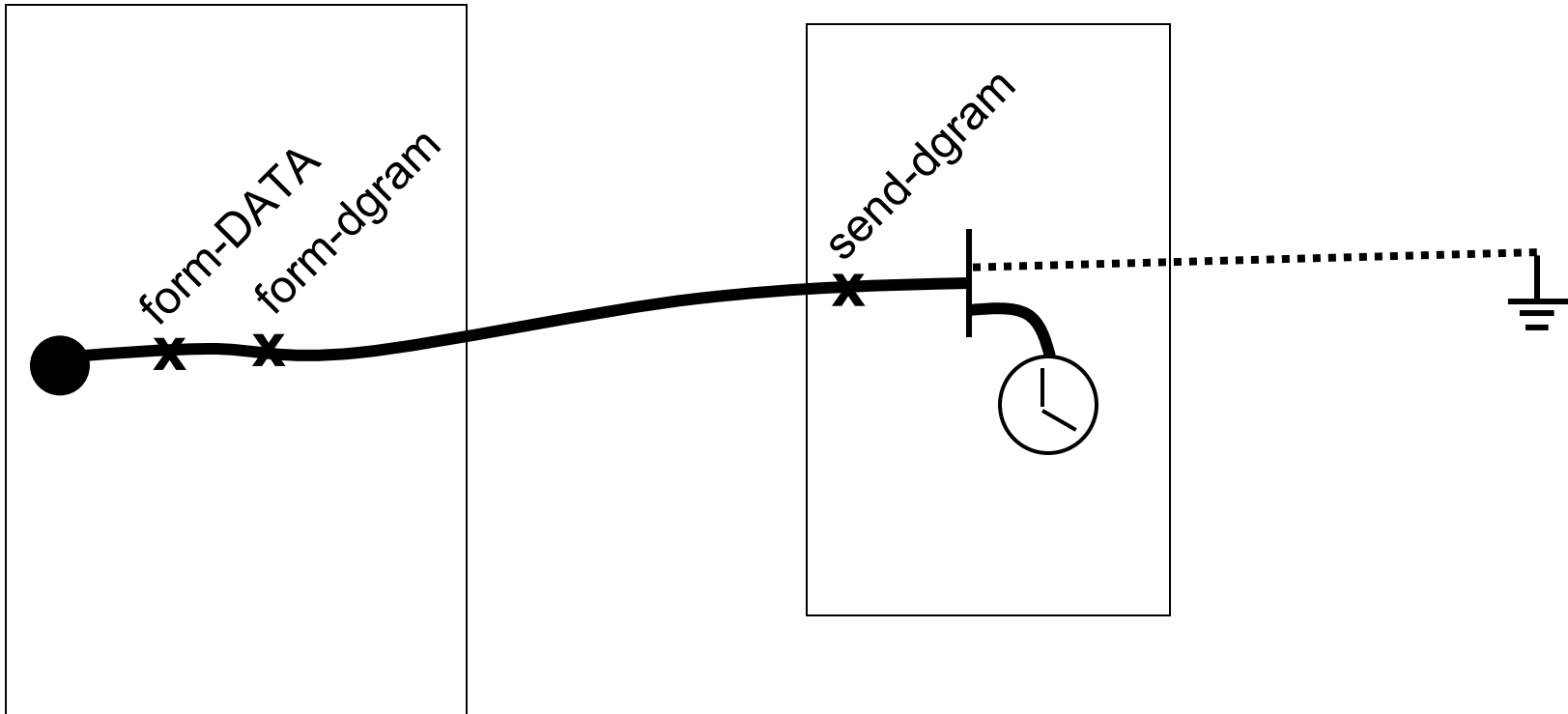
Token Trace: Success Scenario



Token Trace: DATA Packet Lost (1)

Client

DatagramSocket

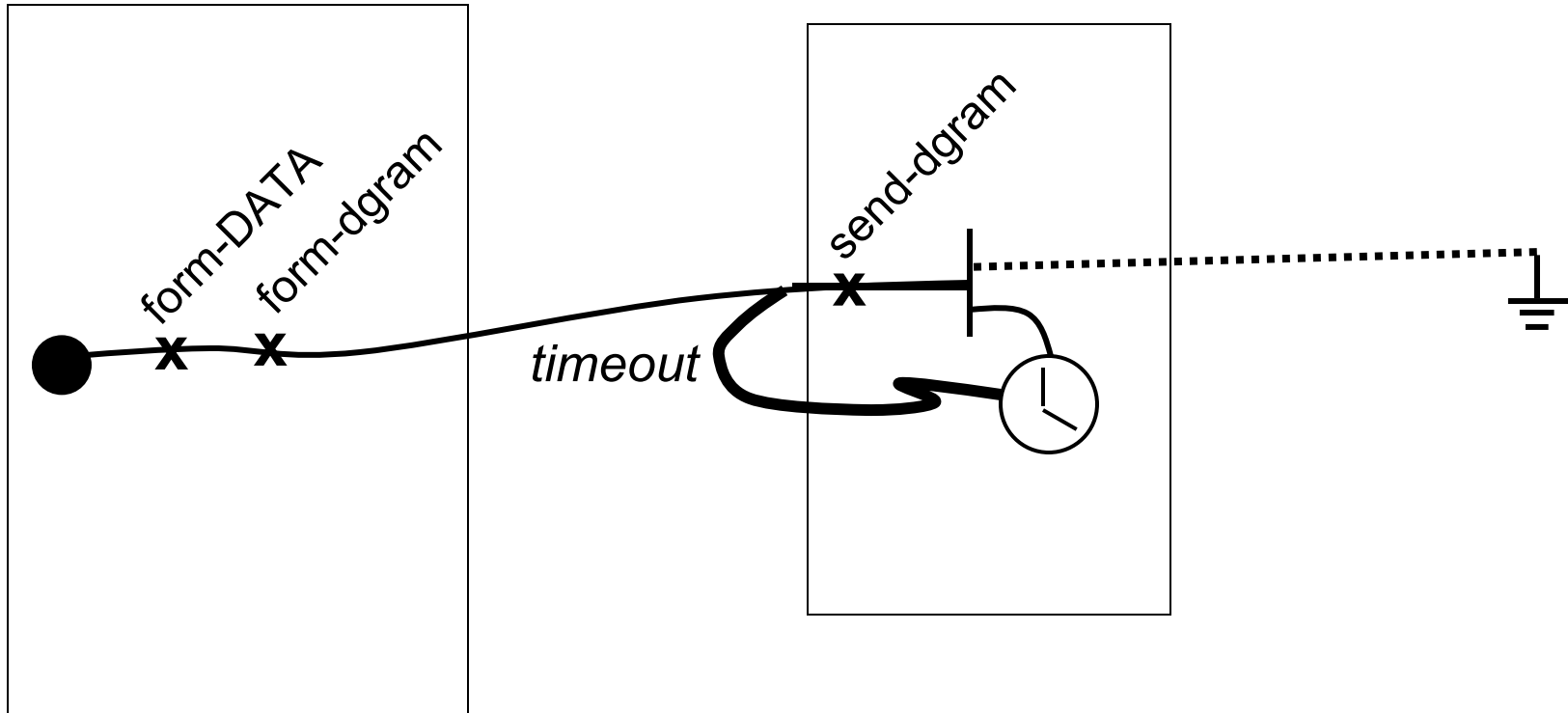


—— First Token Second Token

Token Trace: DATA Packet Lost (2)

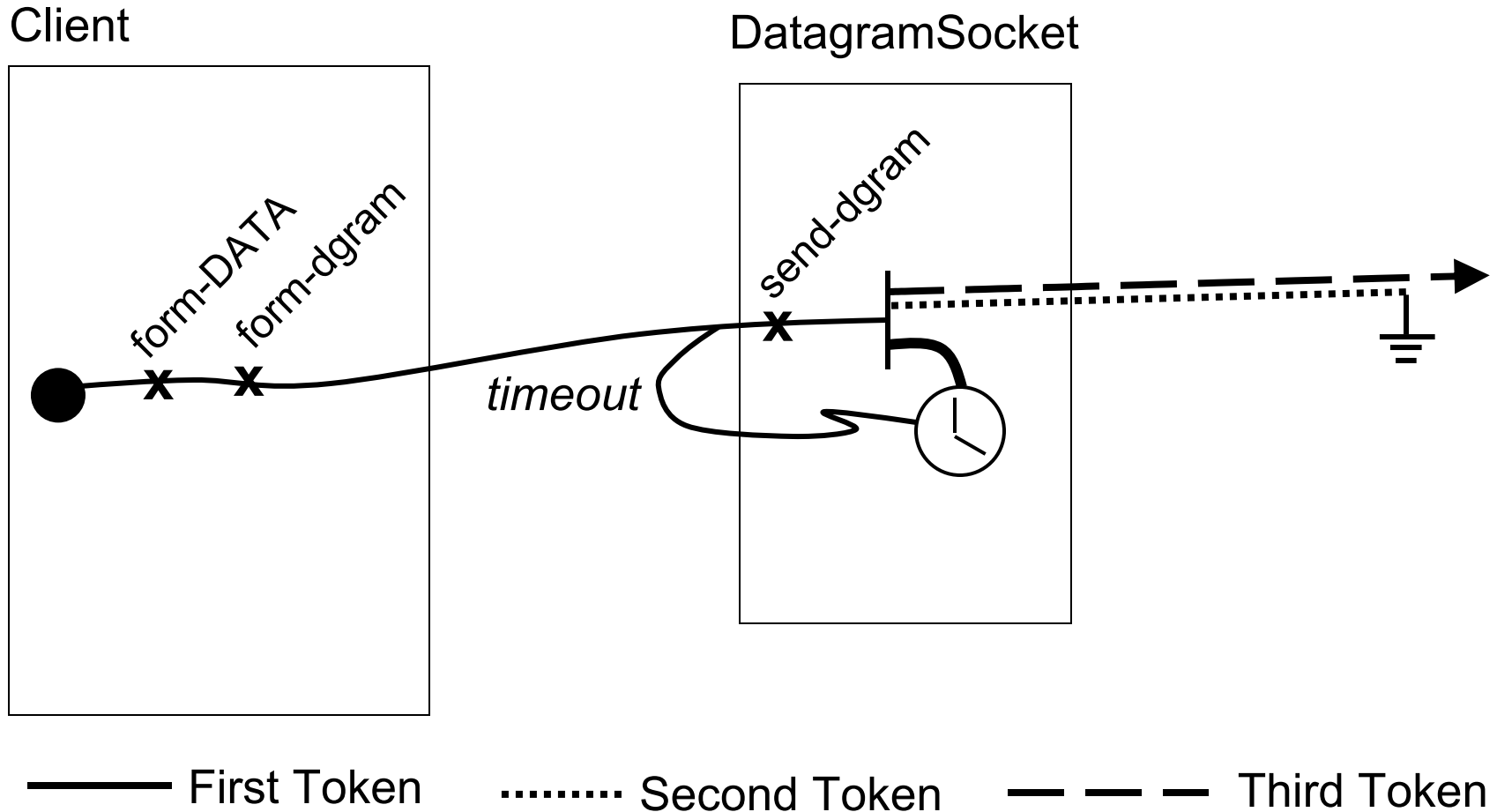
Client

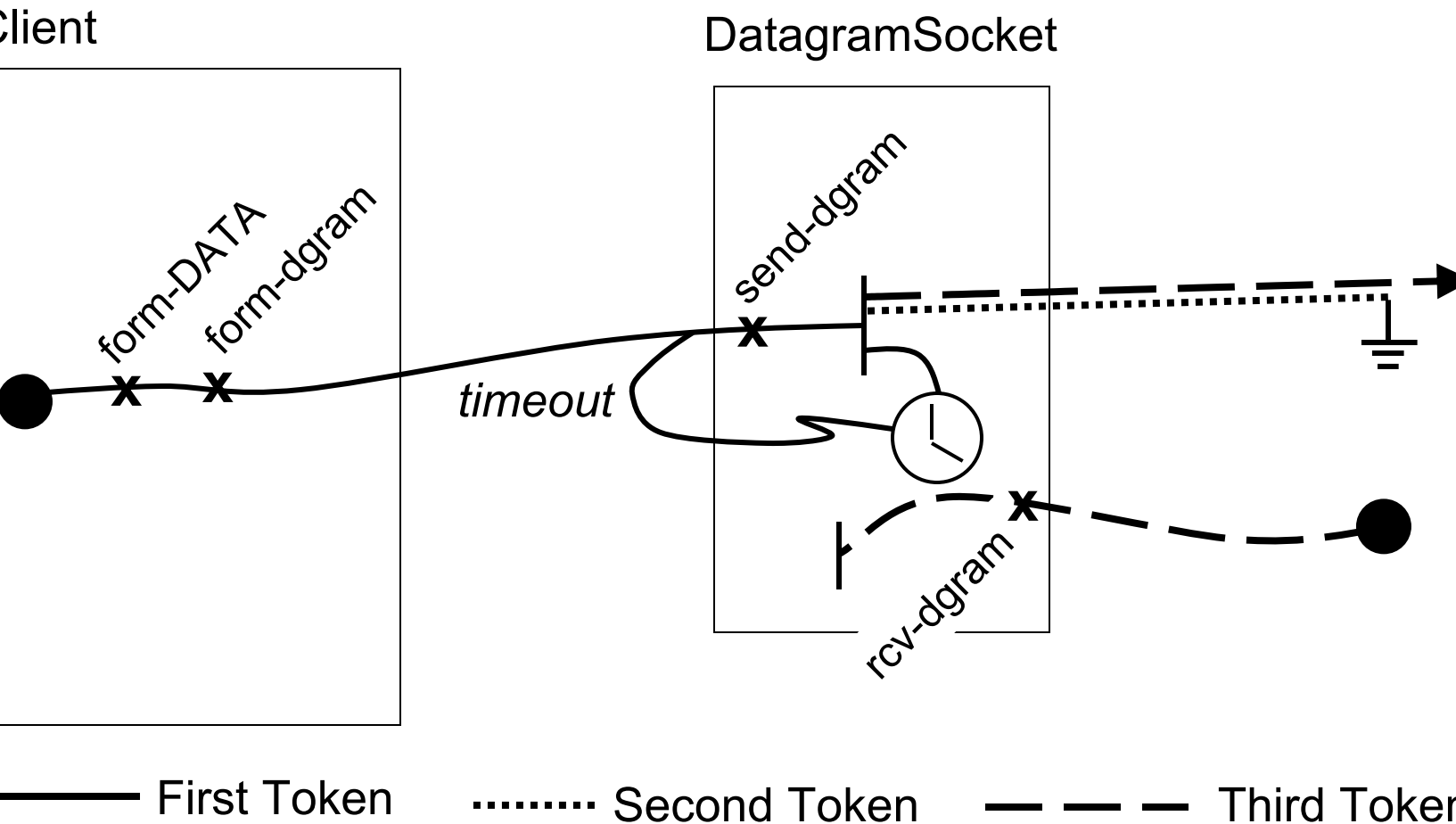
DatagramSocket

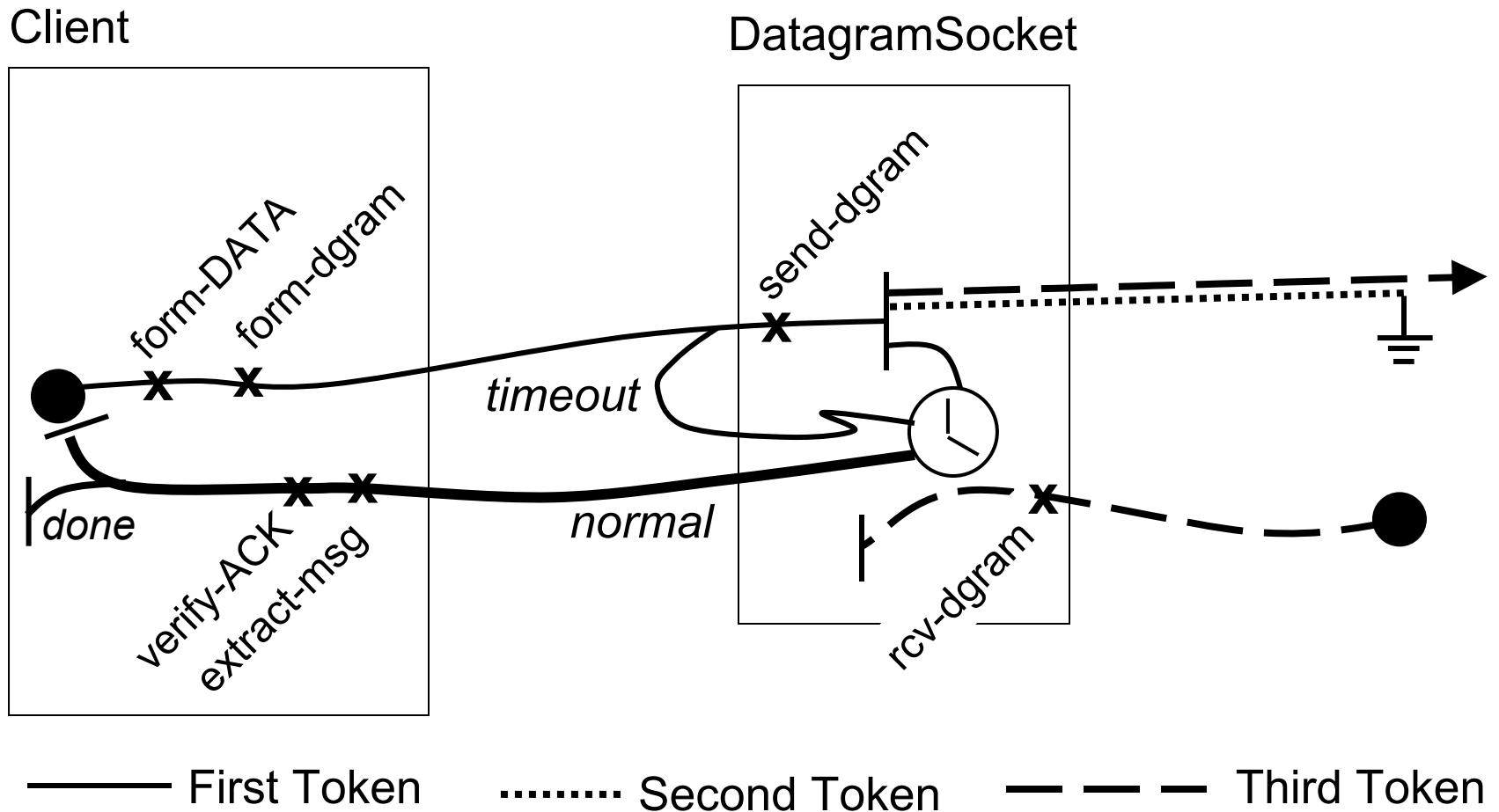


———— First Token Second Token

Token Trace: DATA Packet Lost (3)

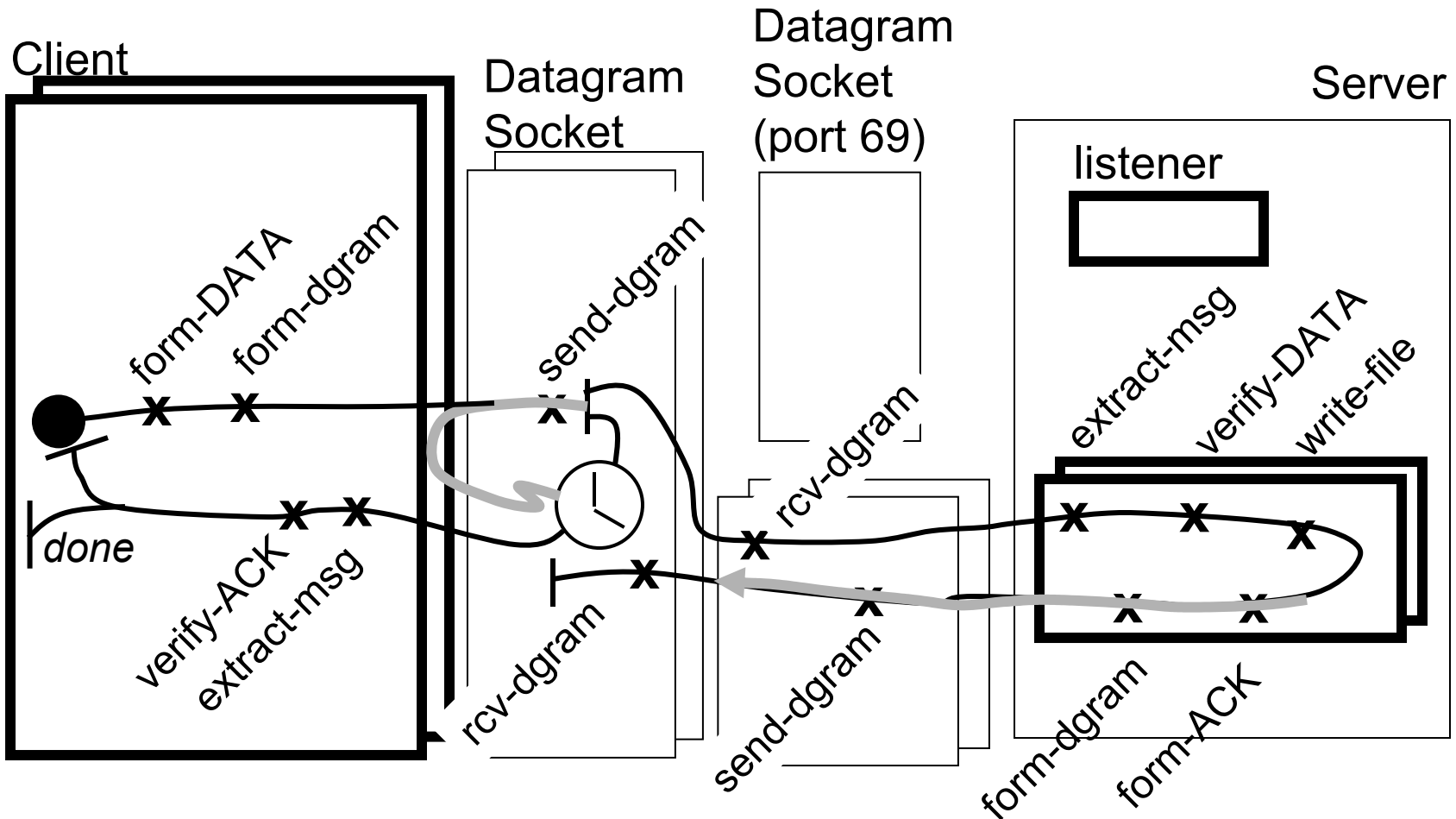






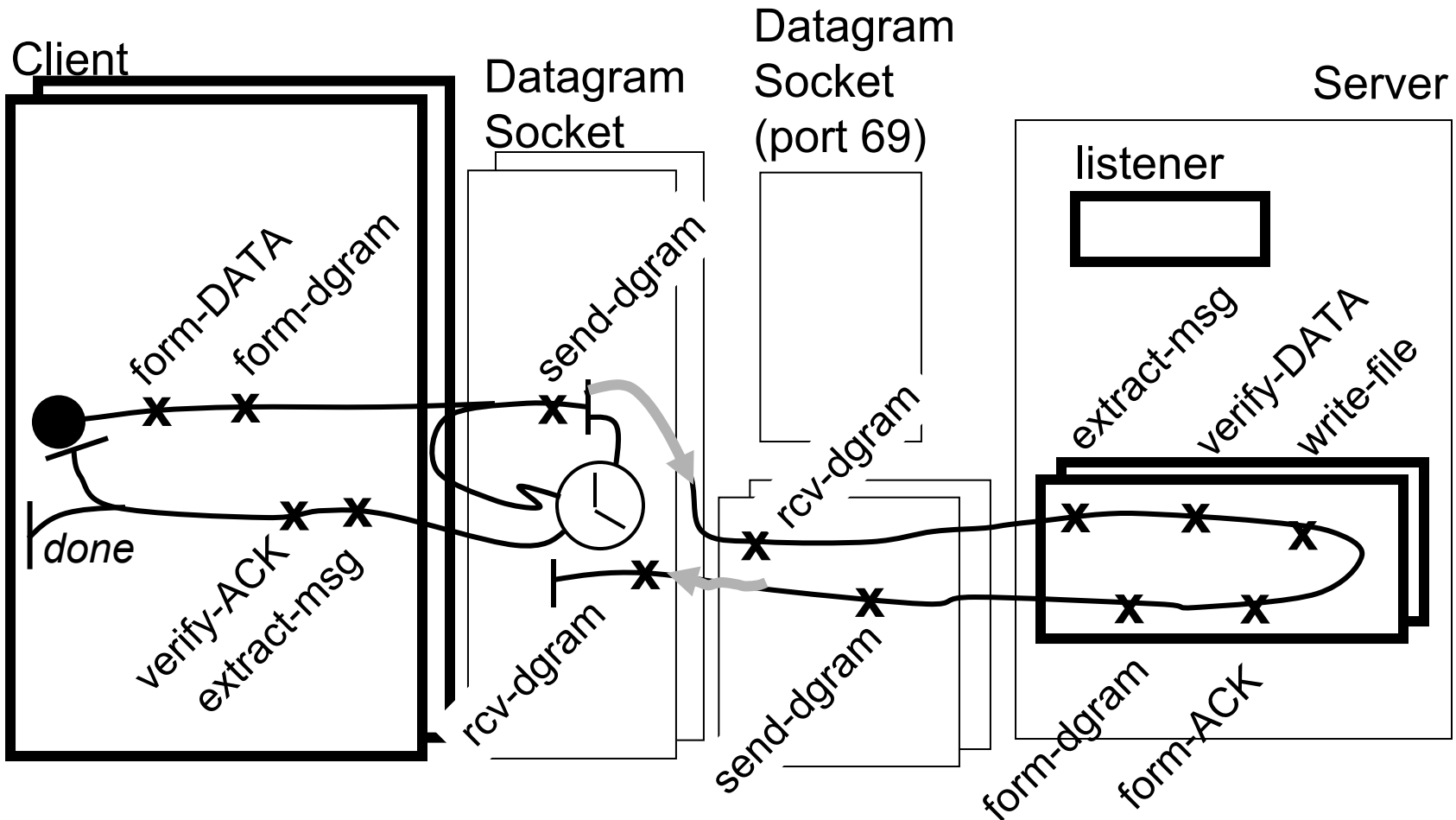
ACK Packet Delayed (1)

Race between timeout and reception of delayed ACK n on client side



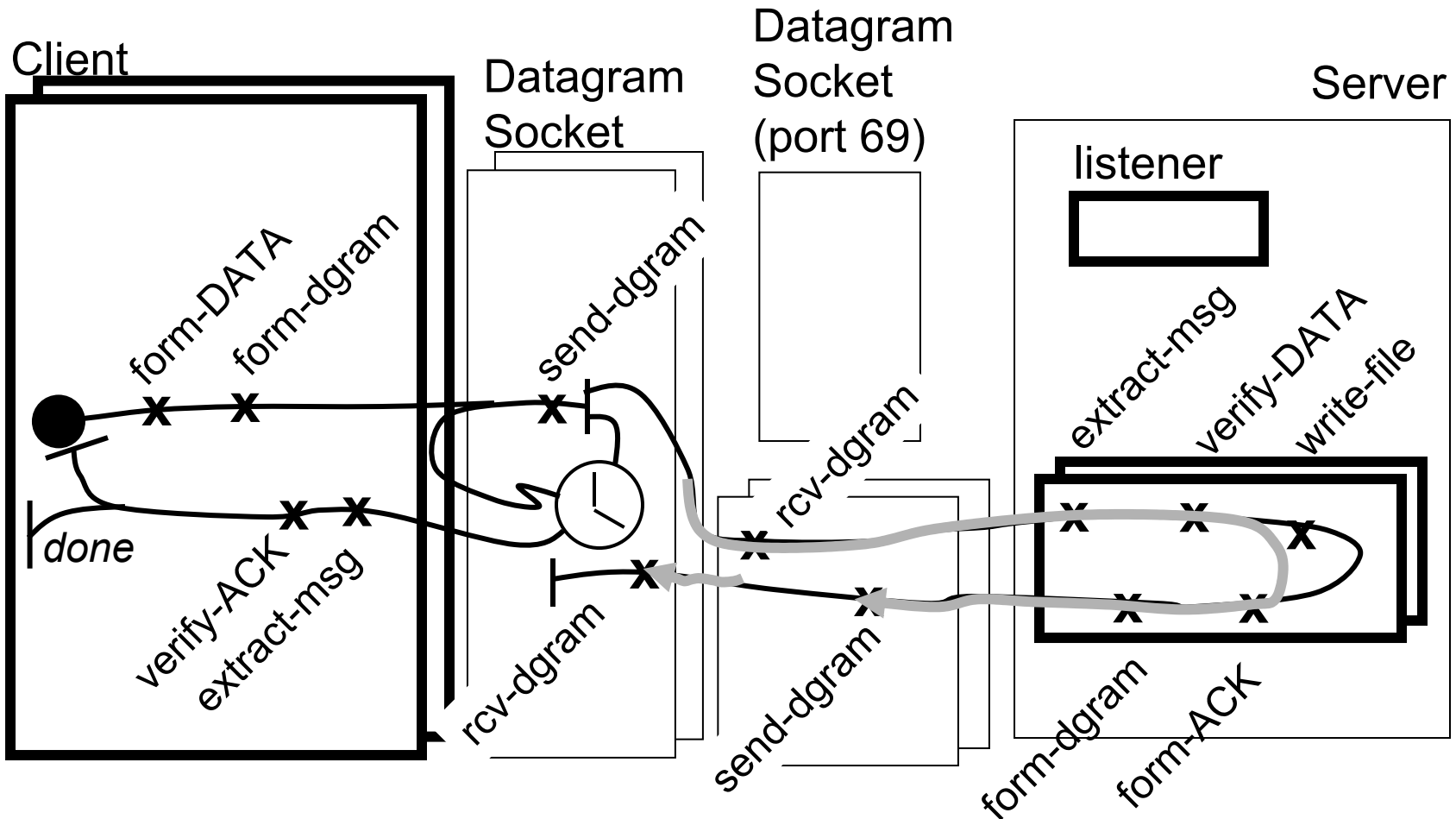
ACK Packet Delayed (2)

Client resends DATA n packet



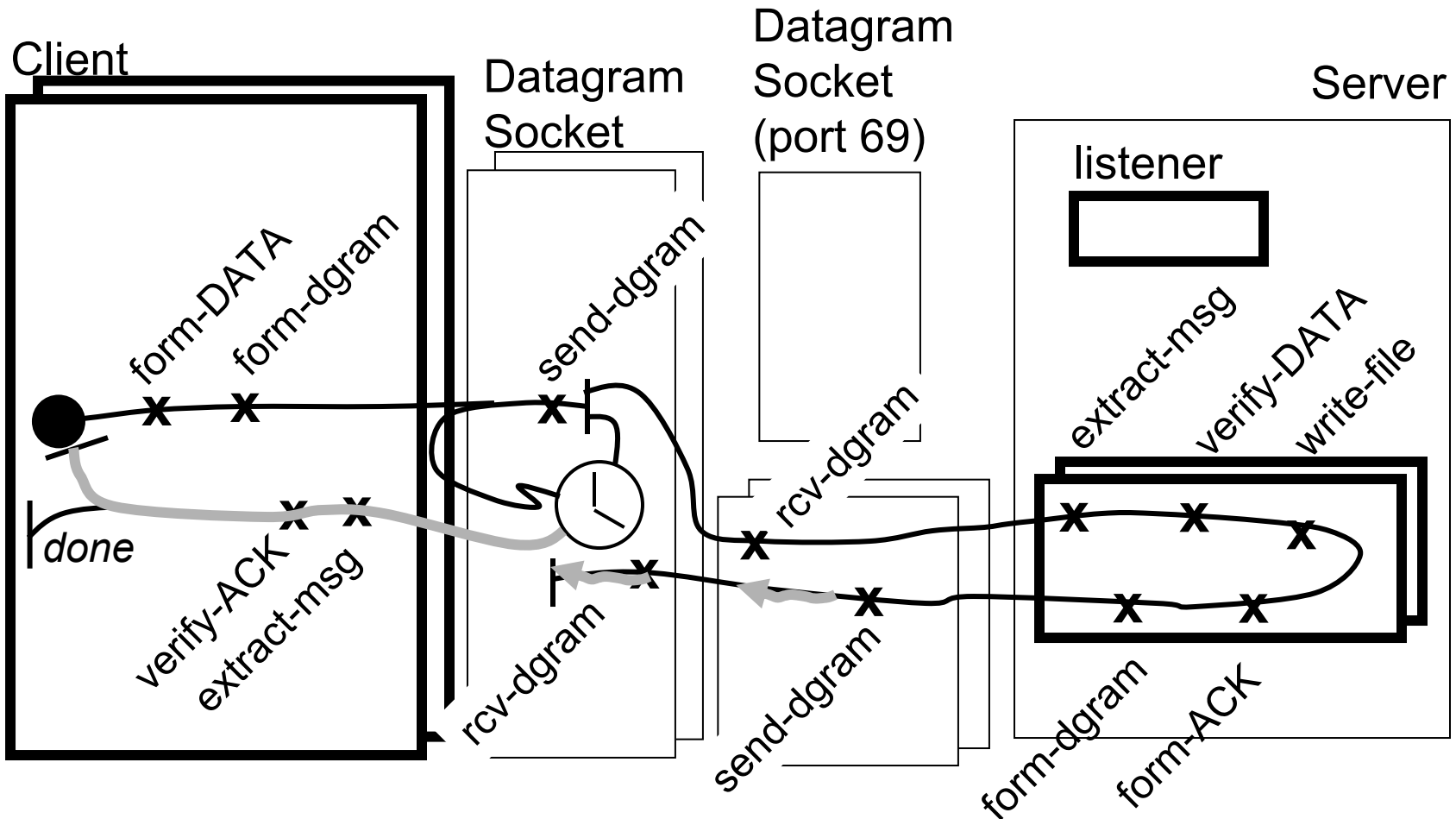
ACK Packet Delayed (3)

Server receives duplicate DATA n packet, discards it, sends ACK n



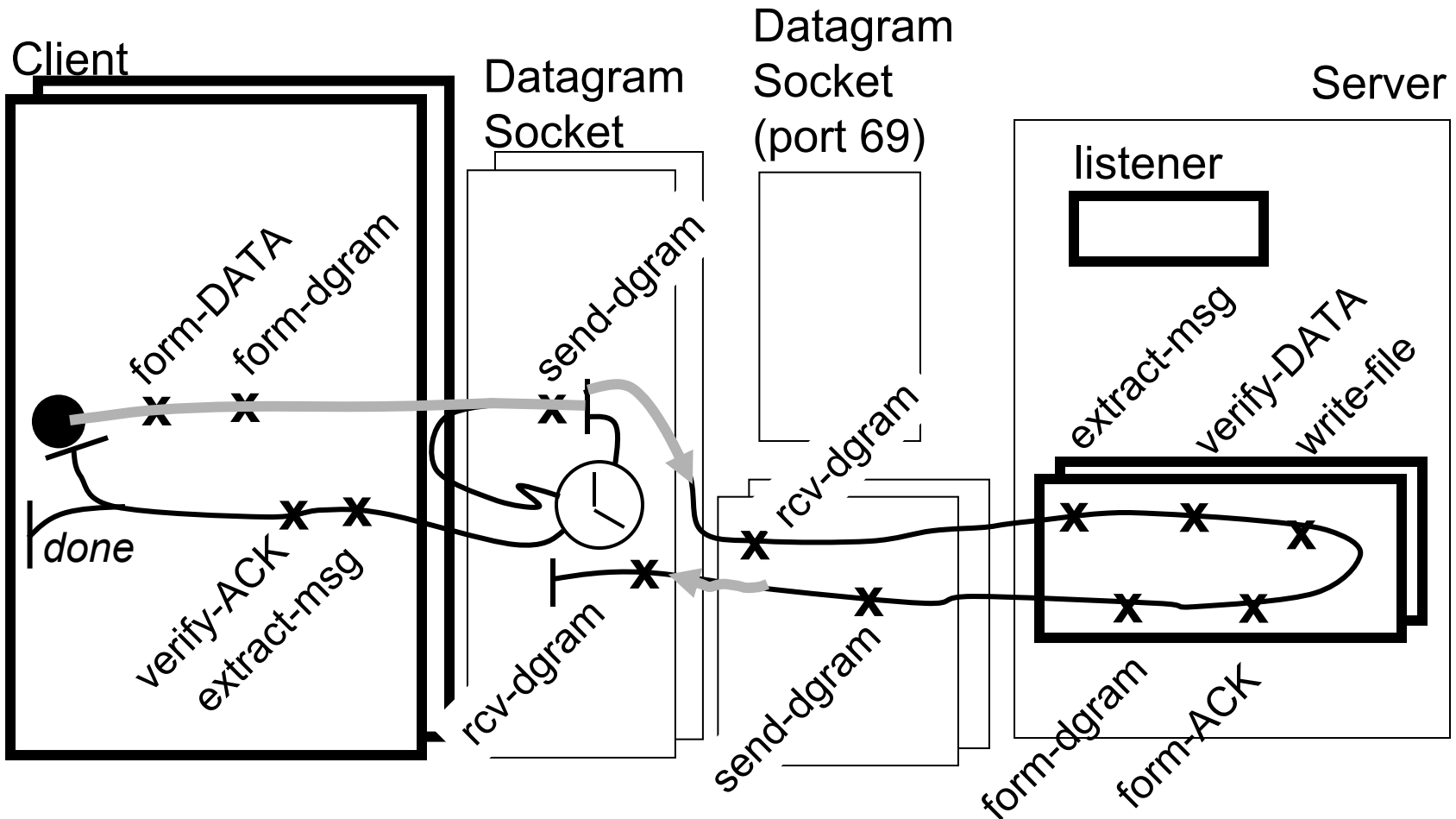
ACK Packet Delayed (4)

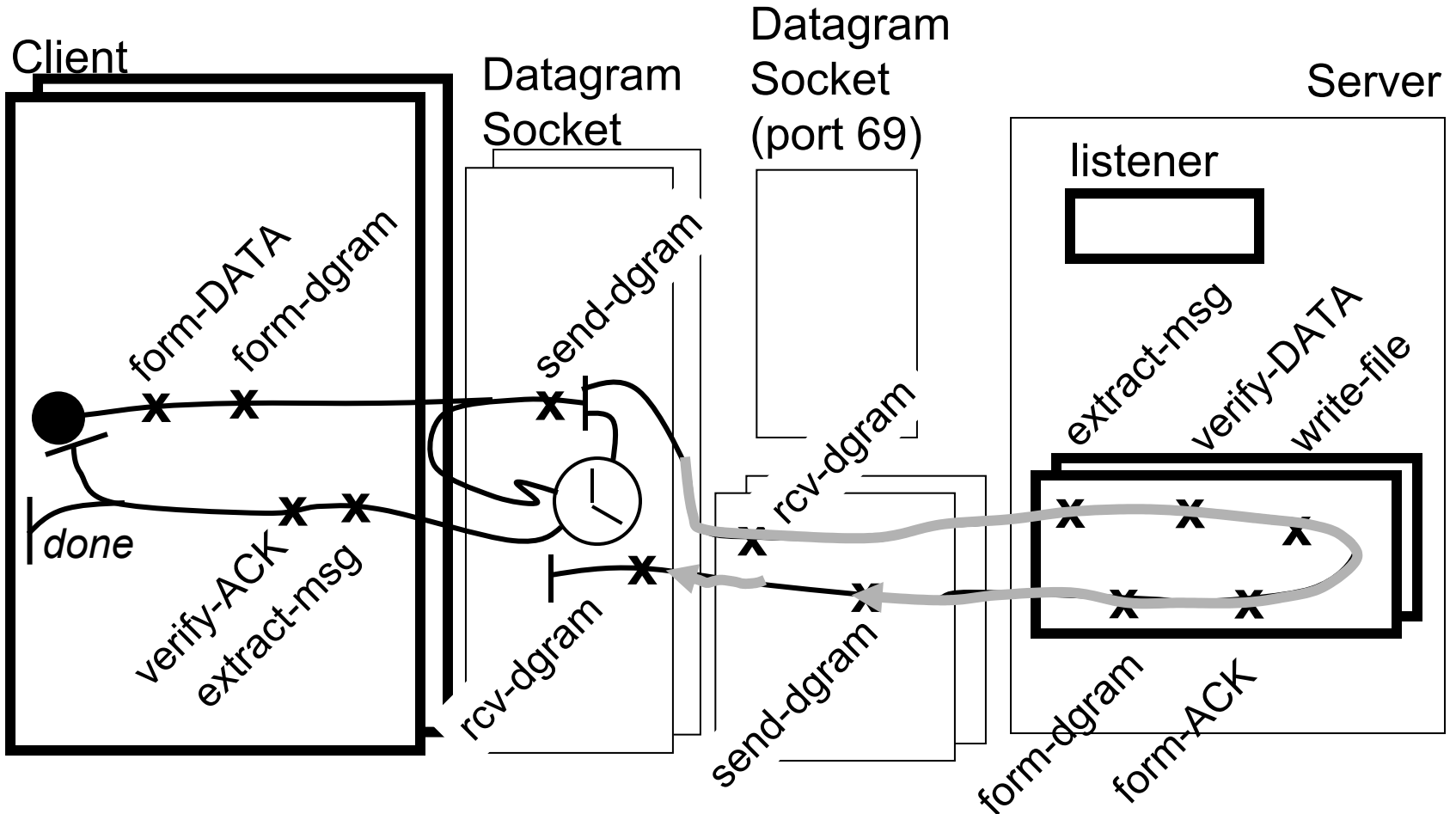
Client receives delayed ACK n packet



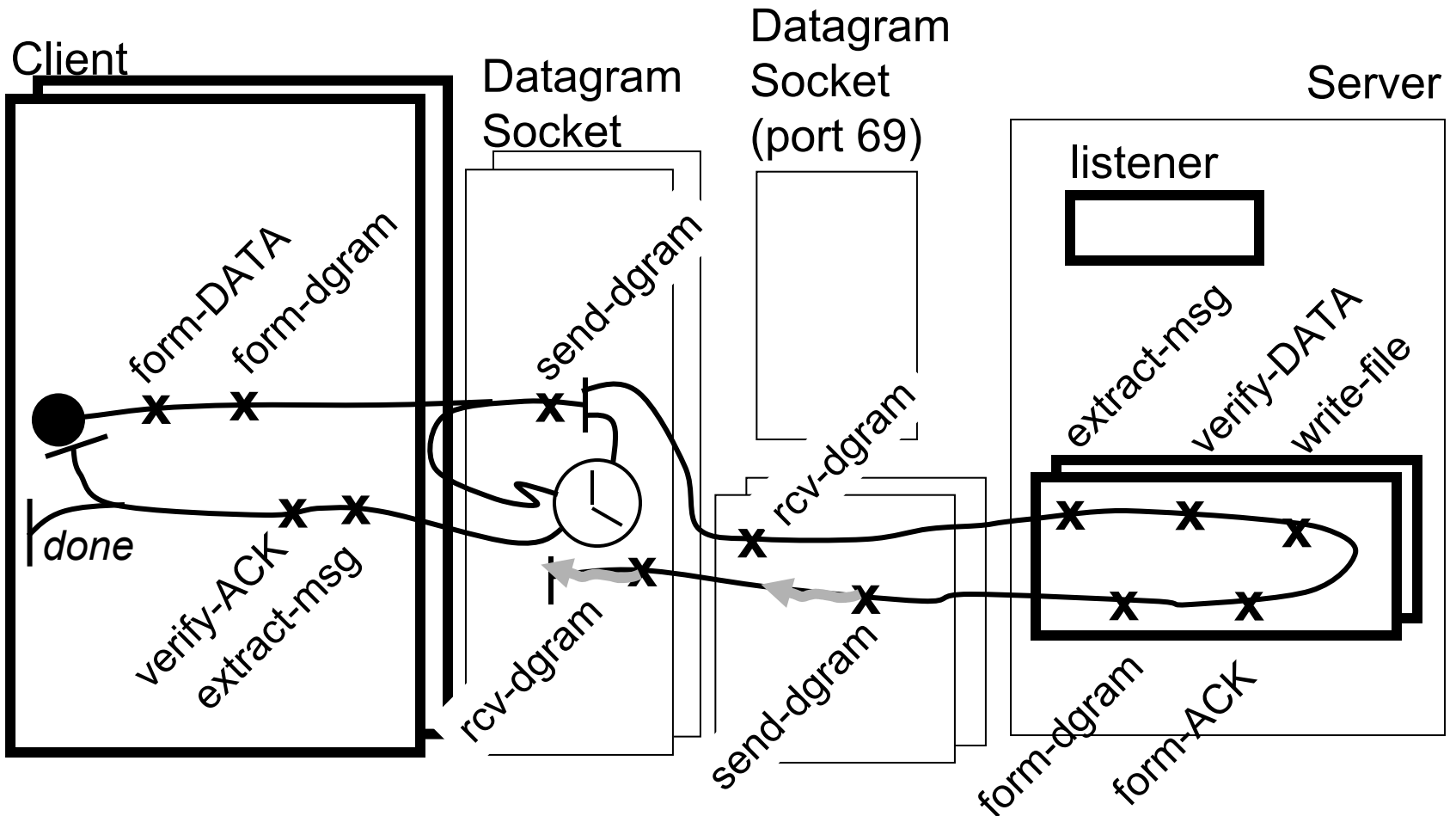
ACK Packet Delayed (5)

Client sends DATA $n+1$





Client receives duplicate ACK n



ACK Packet Delayed (8)

- If the fix for the Sorcerer's Apprentice bug has been implemented, the client will discard the duplicate ACK and wait for the next packet (the ACK $n+1$ packet) (see next slide - note the new path segment)

ACK Packet Delayed (9)

Client discards duplicate ACK n

