

# CmpE 275

---

Section: Decomposition of data  
(streaming, failure detection, and patterns)

Everything as an event...

# Agenda

- Lecture
  - ♦ Discrete event modeling (applied messaging)
  - ♦ Asynchronous
  - ♦ Fail fast strategic design choices
    - Patterns: Proactor, Circuit Breaker, an Heartbeat
- Key points
  - ♦ Workflow/Data decomposition
    - Event ordering
  - ♦ Building QoS and failure detection
  - ♦ Patterns: CB, HB, and BH
  - ♦ Introduction to Netty

## Lab

1. Setup network
2. Protobuf

Next week!  
(I forgot the HW)

# Imagine a line of code...

```
// Given
```

```
Data getData() { return _data; }
```

```
Data d = getData();
```

# Imagine copying a file...

```
1. from = open("afile.txt", "r")  
2. data = from.read()  
3. to = open("bfile.txt", "w")  
4. to.write(data)
```

# Imagine distributed...

- Identify concerns with these two examples

1.

2.

3.

4.

5.

Streaming or event processing is the decomposition of a larger conversation, data, or action into many (N) parts.

# Discrete event modeling

- Simulation of complex systems to understand/predict behavior given a set of initial assumptions/conditions
  - ◆ Uses
    - Load/Stress testing
    - Prediction: Modeling, Financial systems
  - ◆ What we are interested is the application to distributed systems

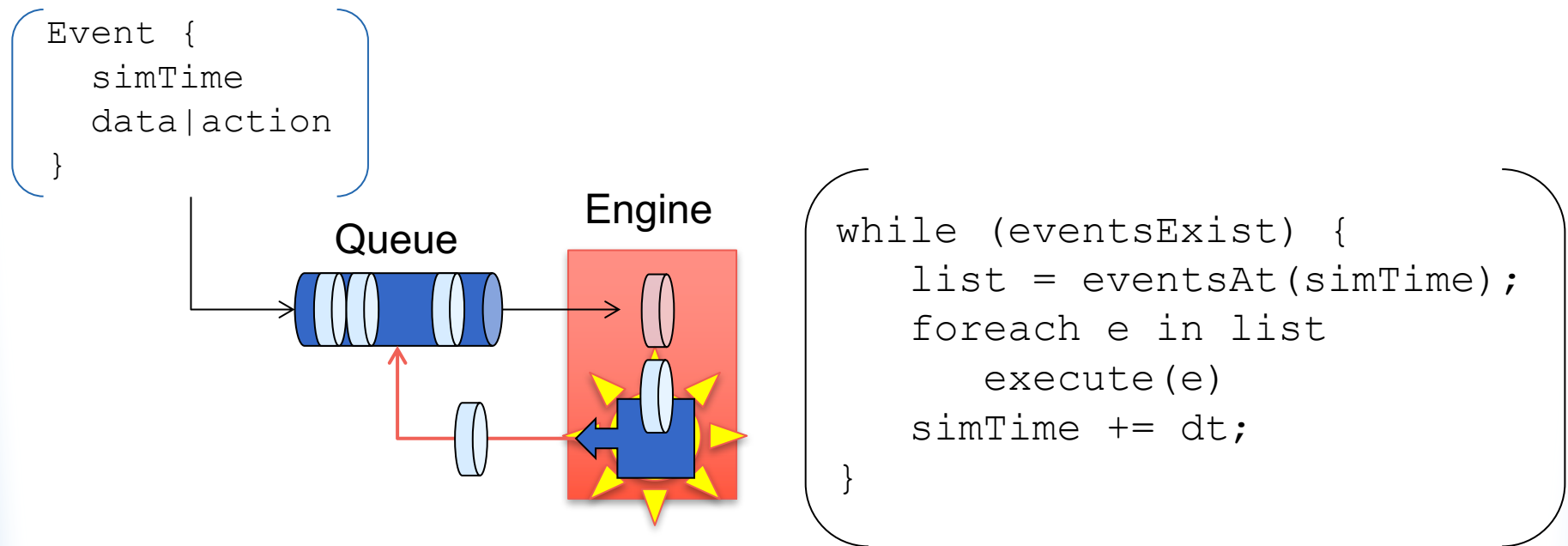
# Types of DES

- Conservative
  - ♦ As events (time) is irreversible. Once execution occurs, it cannot be unwound if new events occur after the current simulation time ( $e.\text{simTime} < \text{engine.simTime}$ )
- Optimistic
  - ♦ Execution is reversible, the engine can unwind executed events to allow events to be injected. This raises the question that all event actions must support an inverse operation (*lossless events:  $4+5 = 10$ ,  $10-5 = 4$* )



# DES: Synchronous events processing as a queue

- The event engine is a queue that is sorted on simulation time
  - ♦ Time is continuous (float) s.t. an one can always insert an event between two events (given  $e1.time \neq e2.time$ )



# DES: Reversing simulation time

- Asynchronous events may arrive to a server after the servers simulation time.
  - ◆ Options
    - Drop the event (usually not a good plan)
    - Rewind simulation time to insert the new event

# DES: Simulation tie breaking options

Choices:

- PRNG (not repeatable – engine must be deterministic)
- Parallel FIFO (in an asynchronous or parallel system ordering is not consistent as there are factors that come into play that the engine has no control over)
- Arrival time ordered (enqueueing is not deterministic)
- Creation ordered
- Natural ordering (an attribute, creation date, ...)
- Delegate to the model

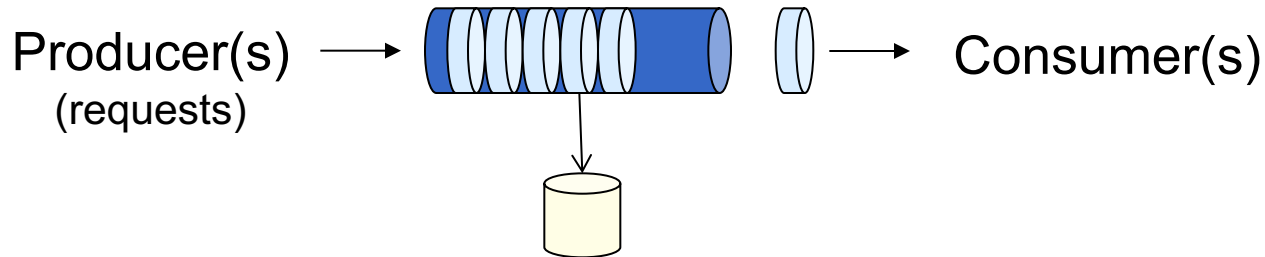
# Back to our line of code...

```
// Given  
Data getData() { return _data; }  
  
Data d = getData();
```

How do we apply DES to getData()?

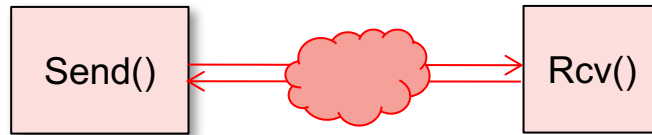
- 1.
- 2.
- 3.
- 4.

# Let's see how DES applies to MQs



- Discrete processing (a.k.a. MQ)
  - ♦ Temporal and Spatial separation
  - ♦ Opaque message payload
  - ♦ Metadata

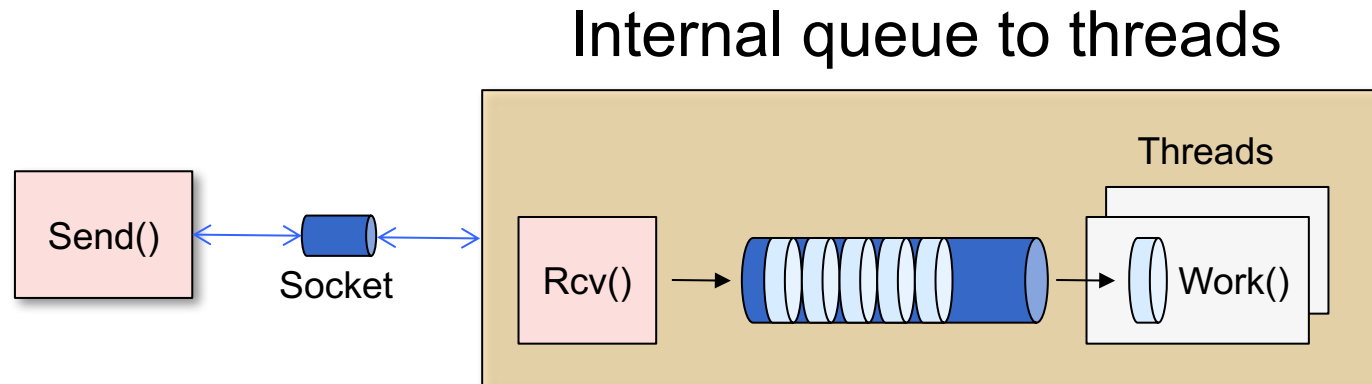
# MQ [DES] Taking a close look at the edges between processes



```
String getName() {  
    return "foo";  
}
```

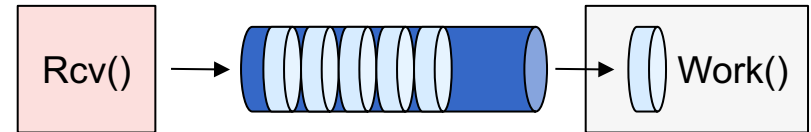
- What is the behavior between the Send() and the Rcv()?
- Synchronous?
- Asynchronous?

# MQ [DES] design applied to load balancing



- What is the behavior of this system under the following conditions?
  - ♦ Idealistic (nominal)
  - ♦ Stressed (over-loaded)

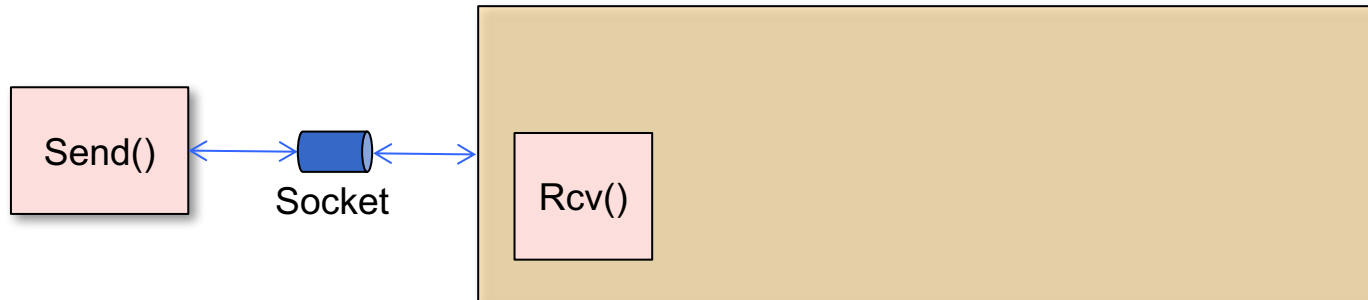
# Looking at the internal relationships



- What is the behavior between `Rcv()` and `Work()`?
- What are the possible strategies when these components are under stress ( `Q >> Q` )?

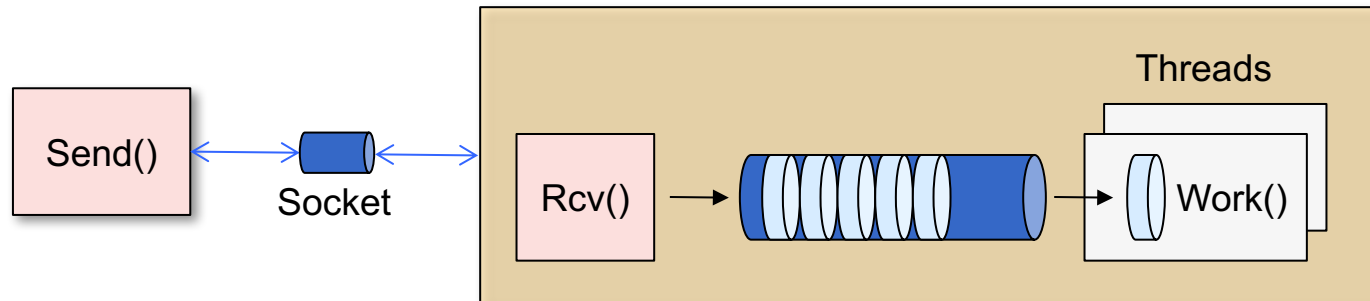


# Looking at the external connection




- What is the behavior between Send–Rcv
  - ♦ Idealistic (nominal)
  - ♦ Stressed (over-loaded)
  - ♦ Threats (SQL injection, Byzantine) and security

# Applying DES to our work

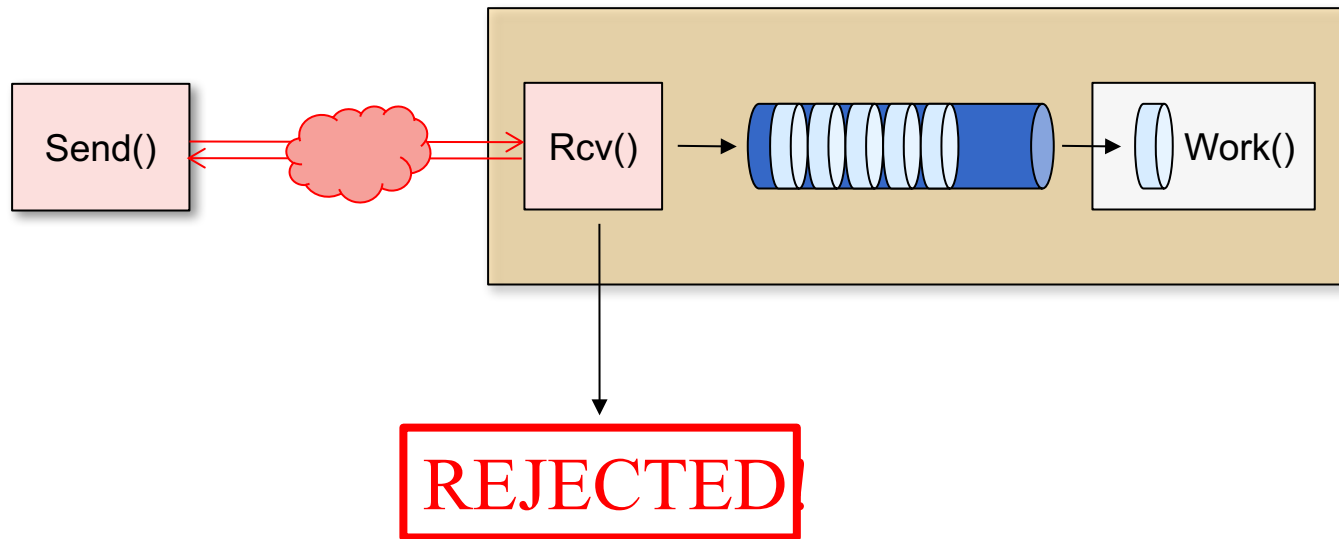


- Enhancements

1. Screen incoming messages before reaching Work()
2. Send() knows if Rcv() is running
3. Rcv() knows if Work() is overloaded or failing
4. Rcv() shares  with other Rcv() processes

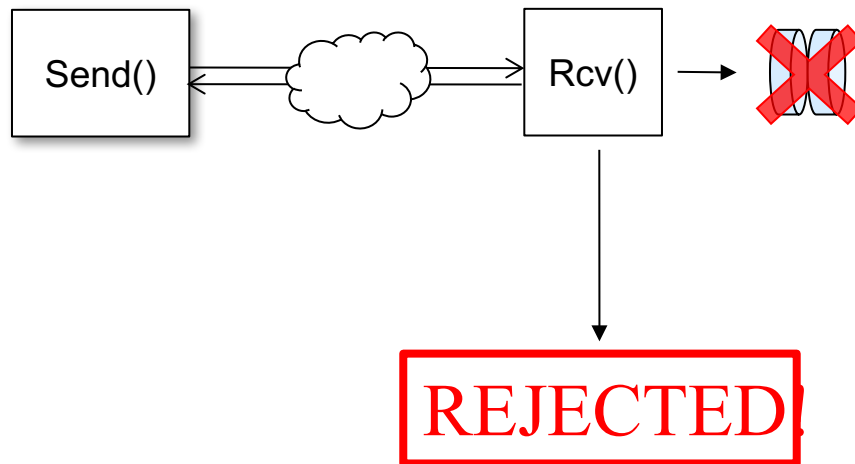
# Queues vs. Fail fast

- Queues allow the server to process requests at its (server) rate unbeknownst to the caller (client)
  - ♦ What if the server did not allow unlimited number of requests?
  - ♦ What if the server rejected the request?



# Aggressive failure

- How can we use failure to our advantage?
  - ♦ Why consider failure if we strive for perfection?

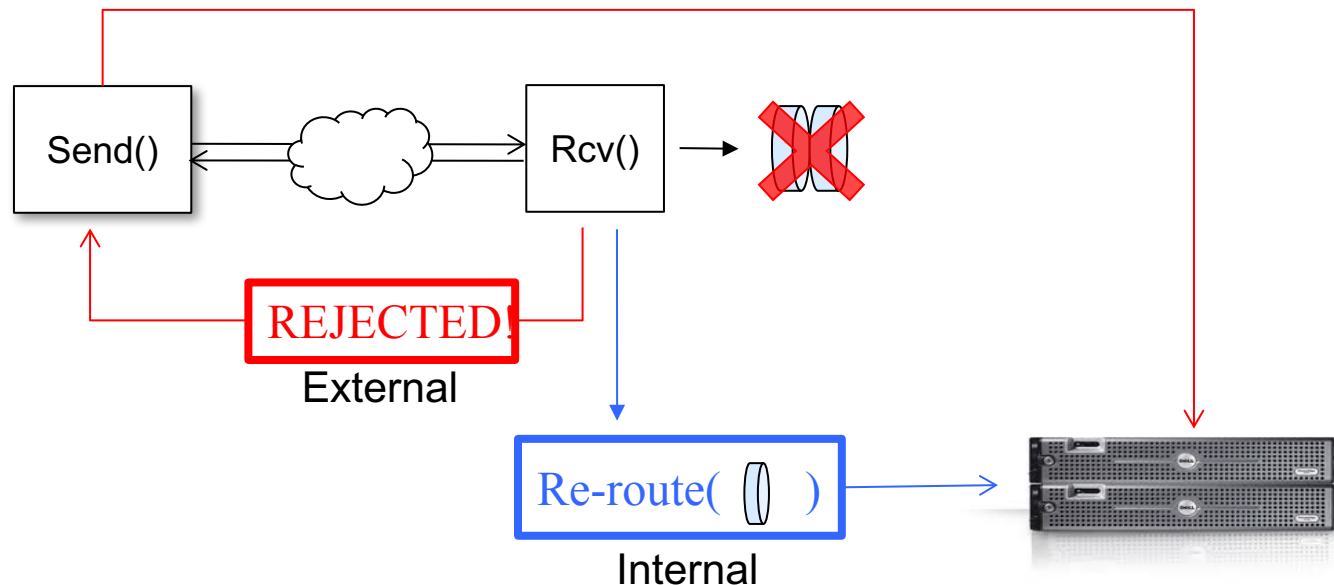


- Failures will occur (Finagles' Law)
- Writing perfect code is hard (\$\$)
- A recovery strategy can complement scaling strategies

# Failure handling (internal vs. external)

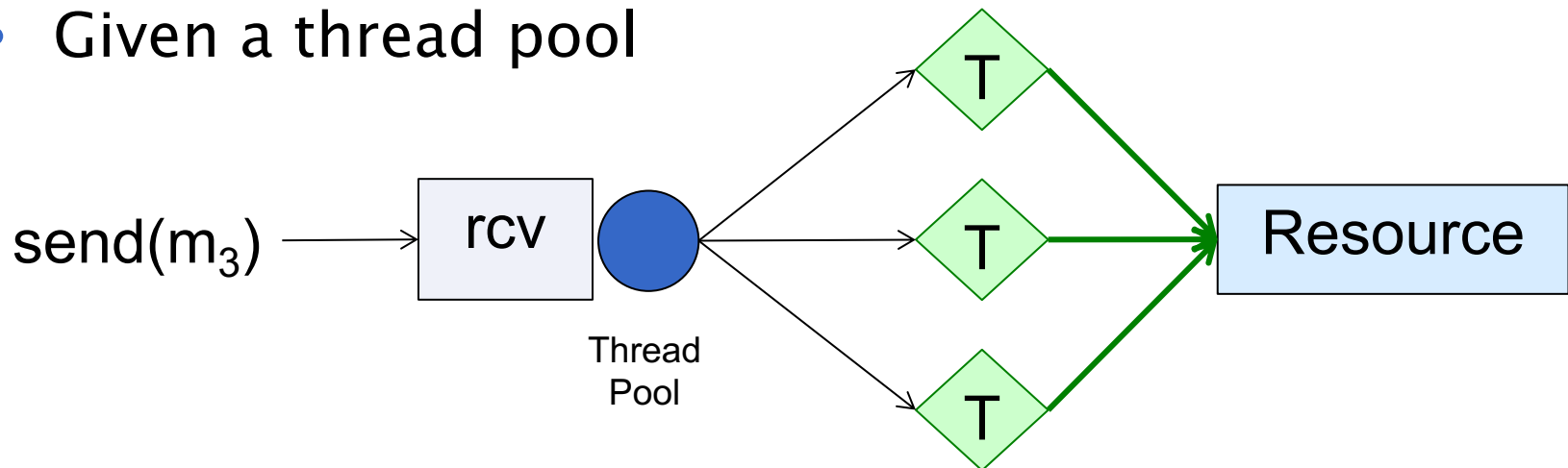
What are the consequences of each choice?

- **Internal:** On failure of a resource, the **Rcv()** can re-route the request (What if **Rcv()** is not reachable?)
- **External:** Throw it back to the requestor (the **Send()** must find another endpoint)



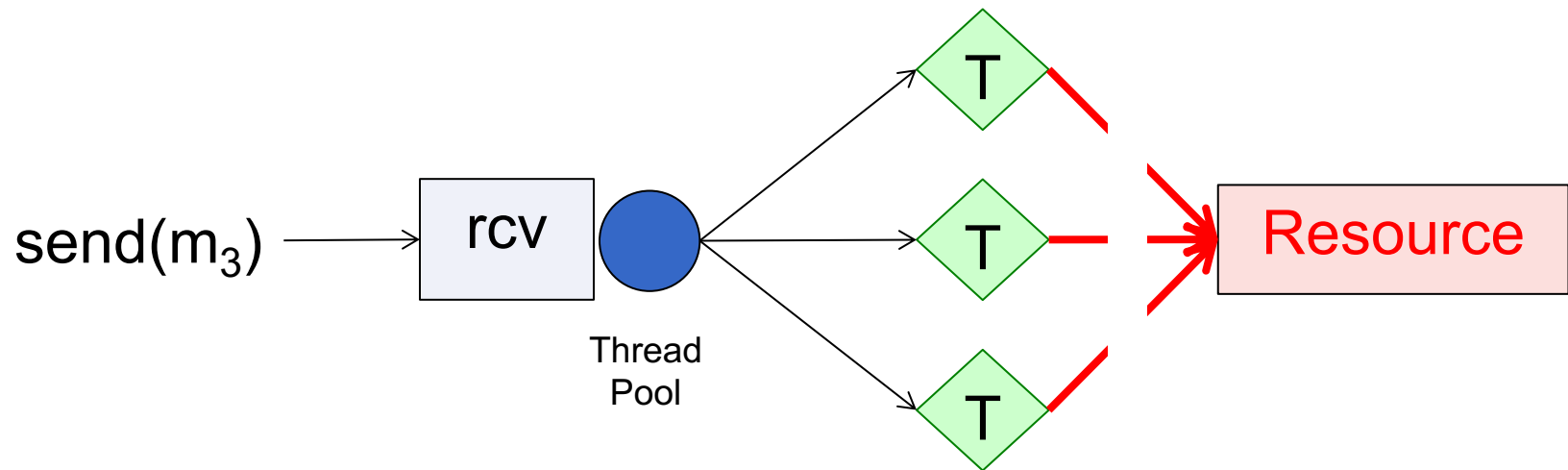
# Pattern: Circuit Breaker

- Circuit Breaker (CB) pattern provides a solution to manage access to resource s.t. error detection (or fault thresholds) can be acted upon once to prevent requests from impacting recovery or retry logic.
- Given a thread pool



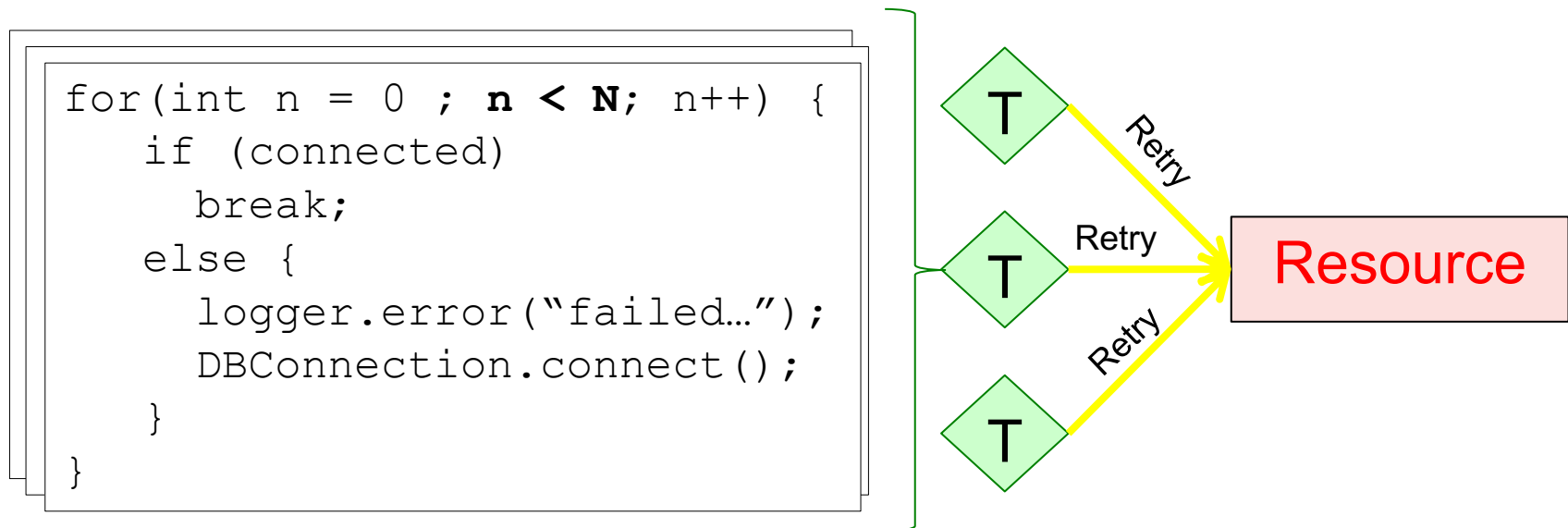
# On failure how do each of the threads (T) respond? The System?

- Let's assume that while the system is running, the resource (or the connection to) fails.
  - ♦ What is the perspective of each thread?
  - ♦ Remember: Isolation of each thread is desired



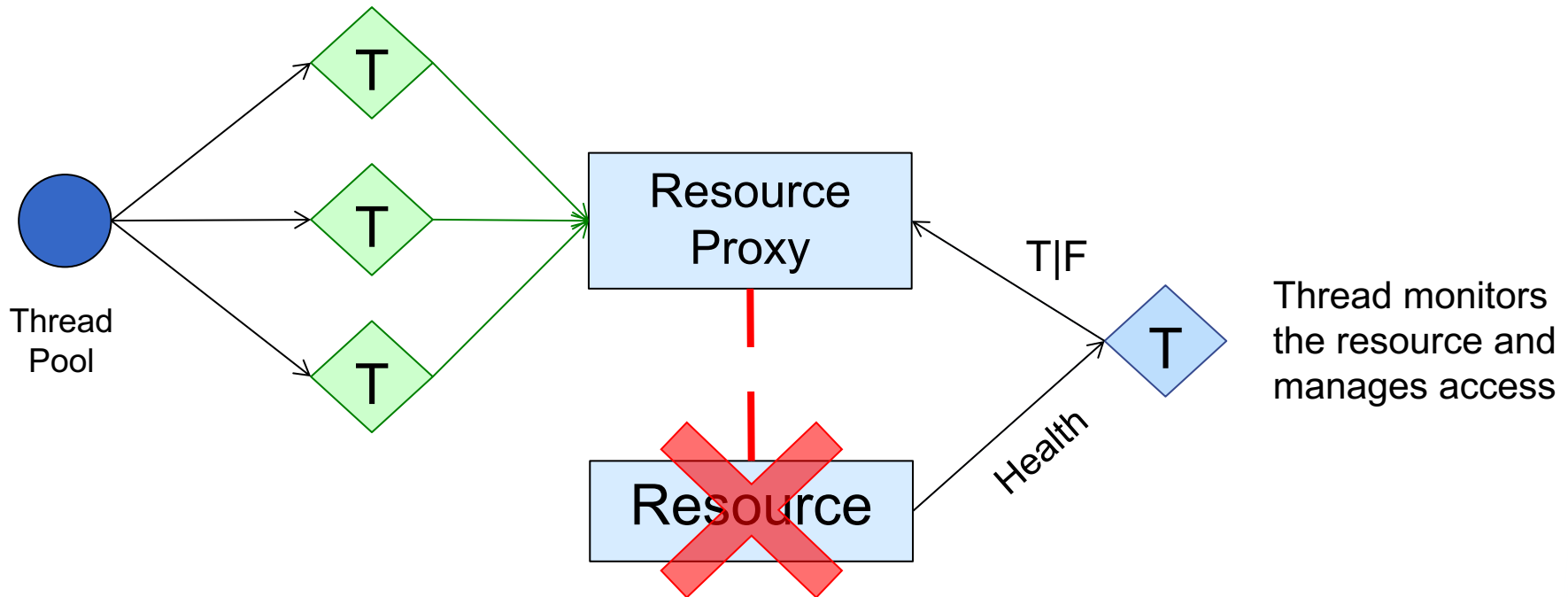
# CB: uncoordinated access to a resource can consume resource cycles, propagate error messages and impact overall performance w/ failure-retry logic

- When a common resource fails, direct access will amplify recovery steps. This can lead to
  - ♦ Increased logging messages ('a failure has occurred')
  - ♦ Performance delays – retry logic can be expensive (e.g., DB connection, sockets)



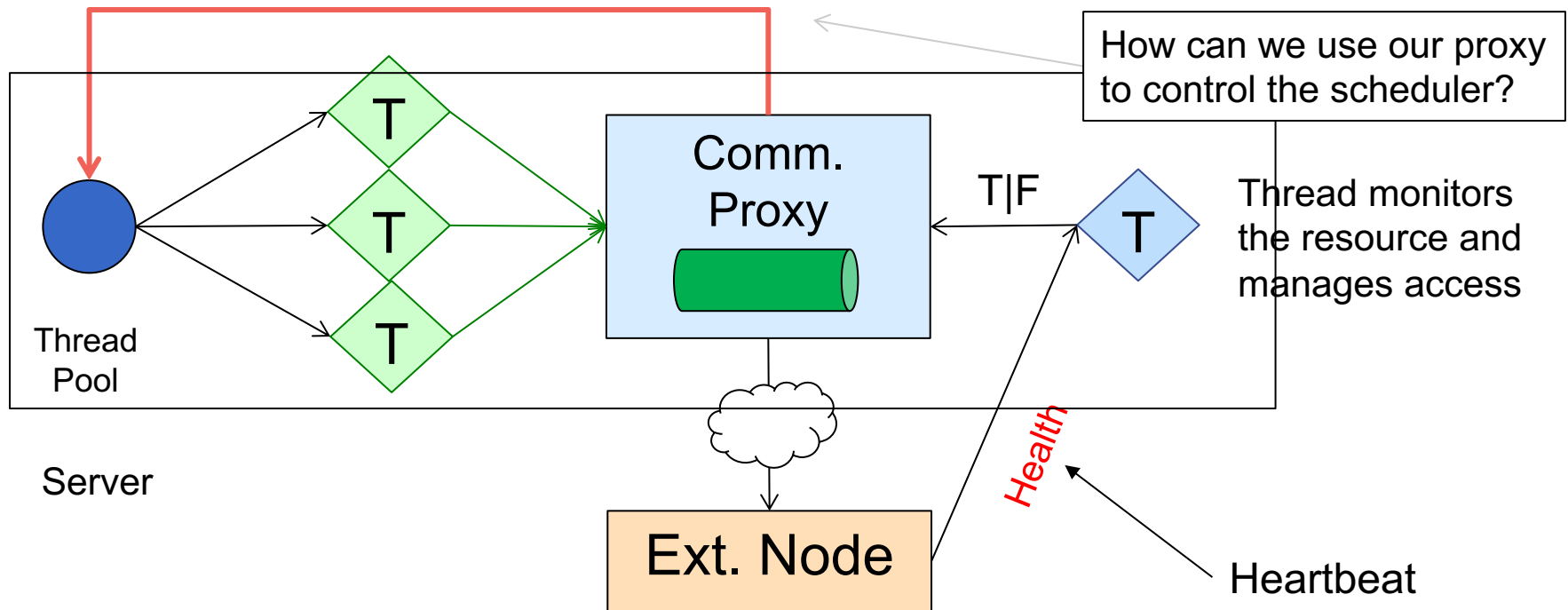


# CB pattern is employed to help reduce the retry and message propagation



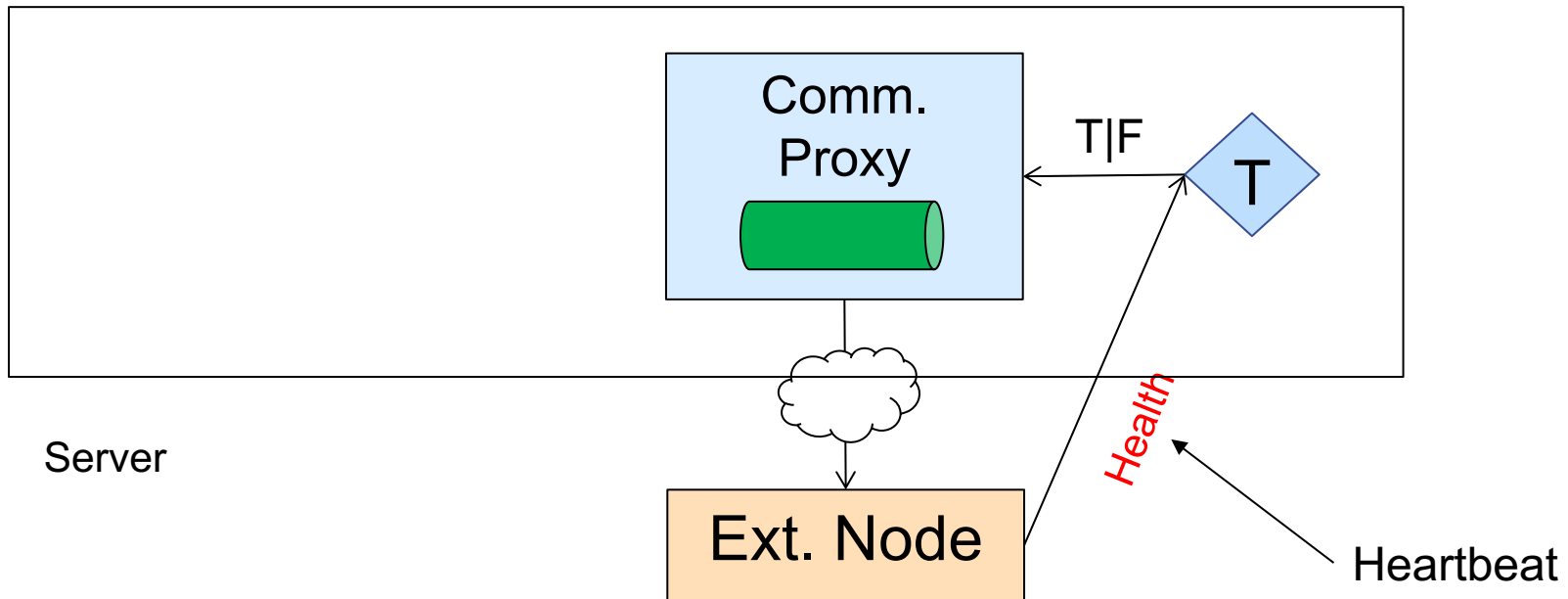
- Monitor thread reacts to a failure or low throughput by shutting off access to the resource – **opens the circuit**
- If the resource is repaired, the proxy is enabled – **circuit is closed**

# Heartbeat applied to monitor internal resources and external services (nodes)



- We have applied several patterns here to create a robust resource manager
- **Patterns: Proxy, Queue, Heartbeat, Circuit Breaker**

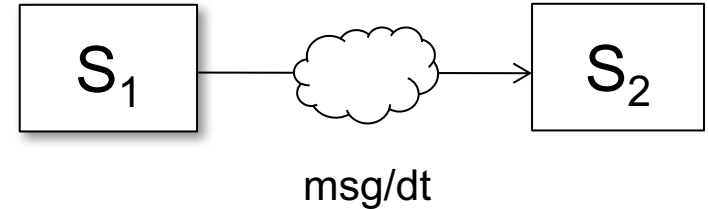
# Heartbeat monitoring of external services (nodes)



- Monitor thread reacts to a failure or low throughput by triggering a restriction to the resource – **opens the circuit**
- If the resource is repaired, the proxy is enabled – **circuit is closed**

# Heartbeat – coding the monitor

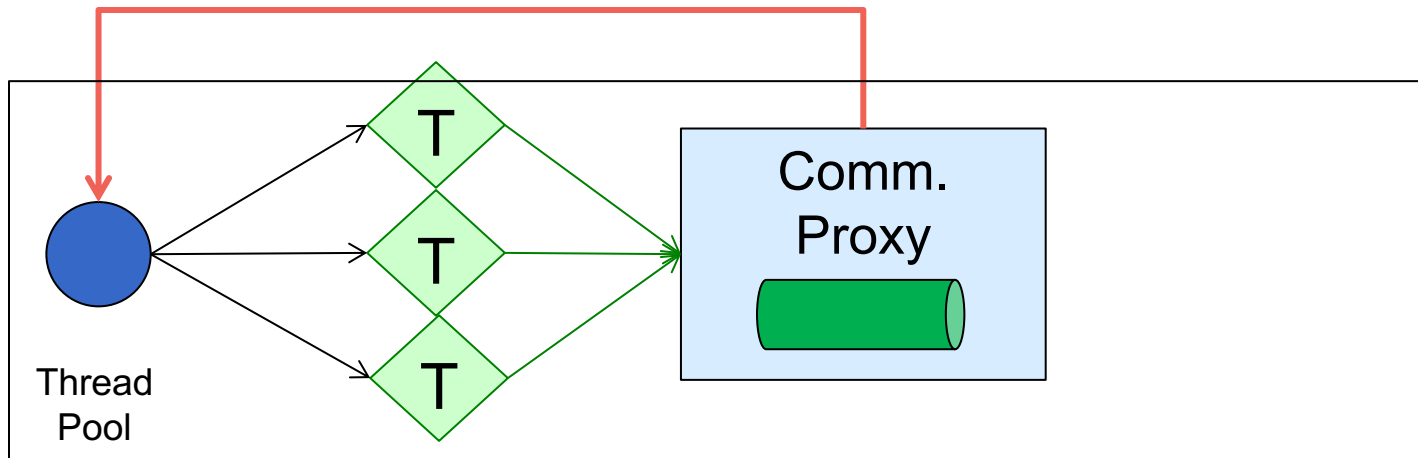
- One-way message
- Lightweight
  - ♦ Small memory overhead
  - ♦ Frequency doesn't impact network
- Detection
  - ♦ Missing sequential messages could indicate a problem



```
long dt = currentTime() - lastMsg(SID).time;
if ( dt > maxTime )
    isolateServer(lastMsg(SID));
else
    lastMsg(SID) = msg;
```

Is this a good design?

# Monitoring internal resources – adaptive QoS algorithms



Server

- Manage the relationship between thread pool size and proxy message pressure can allow or restrict use of the resource Proxy manages (in this case access to an external resource)
- **What is the purpose of the queue in size of the proxy?**
- **How would you design the Thread Pool?**

# Lab work for this week

- Explore netty and the code base
  - ◆ Circuit breaker pattern between nodes
  - ◆ Heartbeat between nodes
    - Can you make the algo more aggressive, less aggressive?
  - ◆ Scale message traffic – How would you add QoS behavior? Where?
- Ordered message processing
  - ◆ We are async. What happens when we must order messages?

# Other Patterns: Bulkhead

- Bulkhead protects the system against resource failures causing cascading problems to the system
  - ◆ hardware isolation and redundancy
  - ◆ Thread management – thread pools
    - Isolate/Reserve threads for management/monitoring use vs. runtime use
      - this ensures that if a problem occurs with the public thread pool, the mgmt pool can monitor and ‘fix’ the public pool

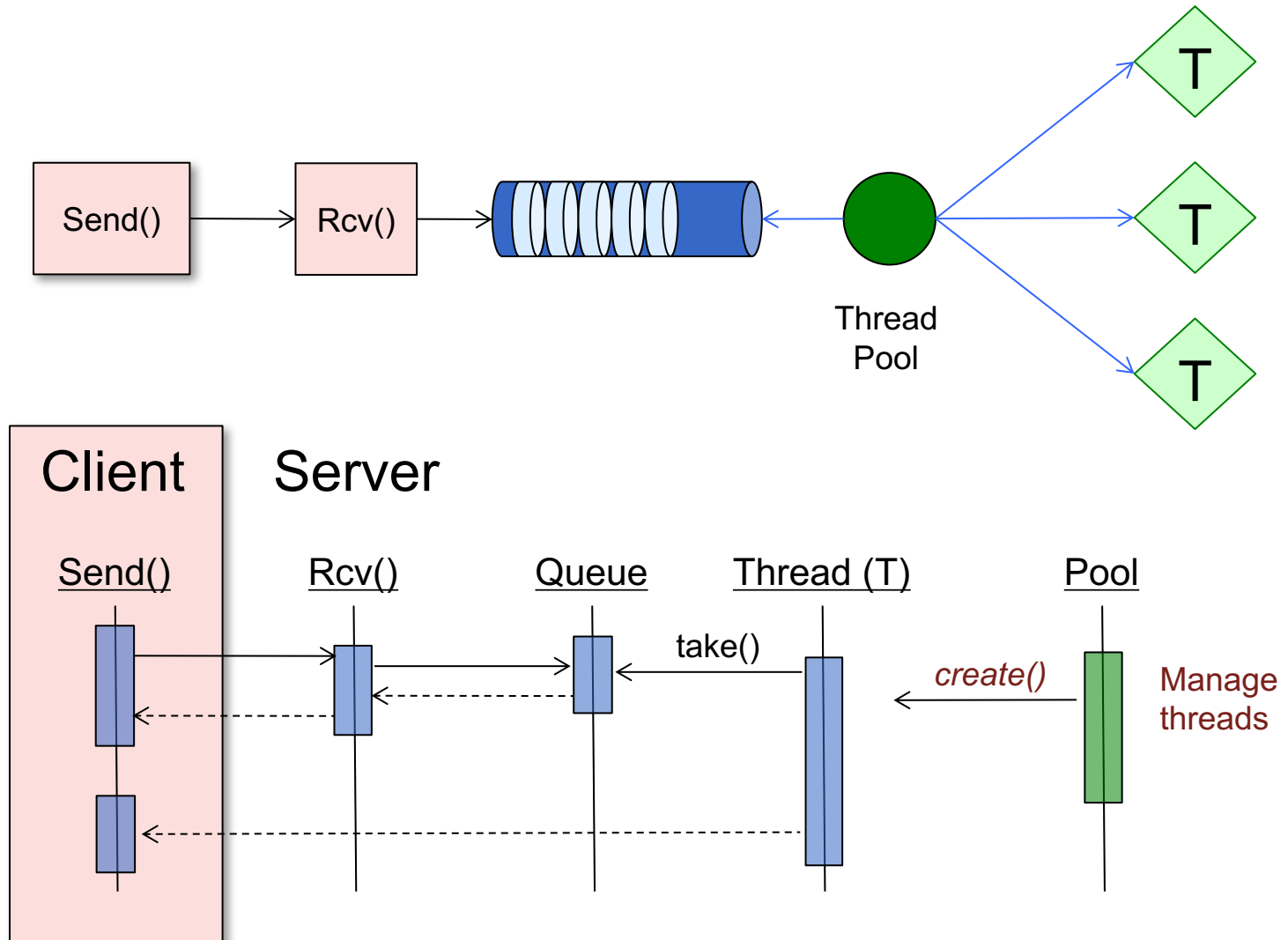
# Pattern: Proactor

- Asynchronous event handling
  - ◆ Queues
  - ◆ Thread pools

How do we arrange a queue and thread pool to provide support for capacity handling?

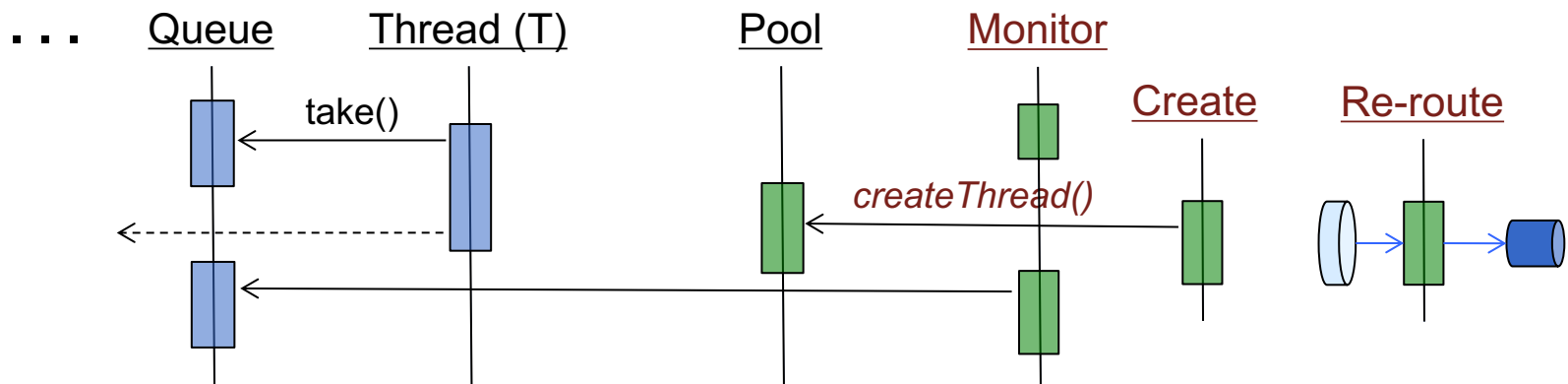
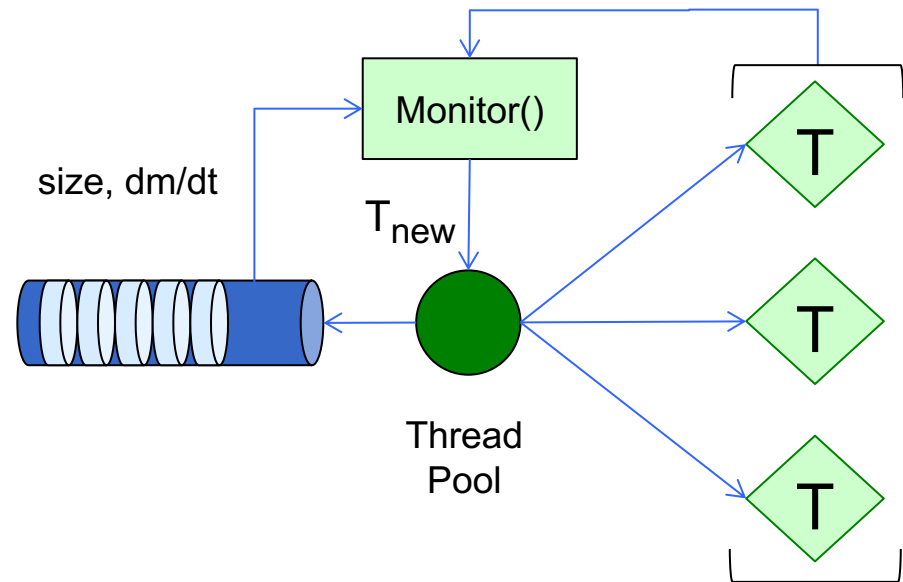


# Proactor combines queues and thread pools to provide asynchronous message processing/scaling



# A Proactor design with feedback for adaptive queue management

- Monitor & Manage
  - ◆ Pool size
  - ◆ Re-route requests
  - ◆ Logging
  - ◆ Manage queue (QoS)
    - Circuit Breaker
    - Prioritize
    - Throttling



# Revisiting the asynchronous get Data

Data getData();

Data d = getData();  
If ( d != null ) process(d);

What if we decomposed the problem into smaller steps?

# The Netty base code

- The Netty base code provides you with an opportunity to look at, explore, and try distributed system concepts.
  - ◆ Queuing behavior
  - ◆ Consensus and election
  - ◆ Qos
  - ◆ Multi-language systems
  - ◆ Standards building
  - ◆ Configuration and code management

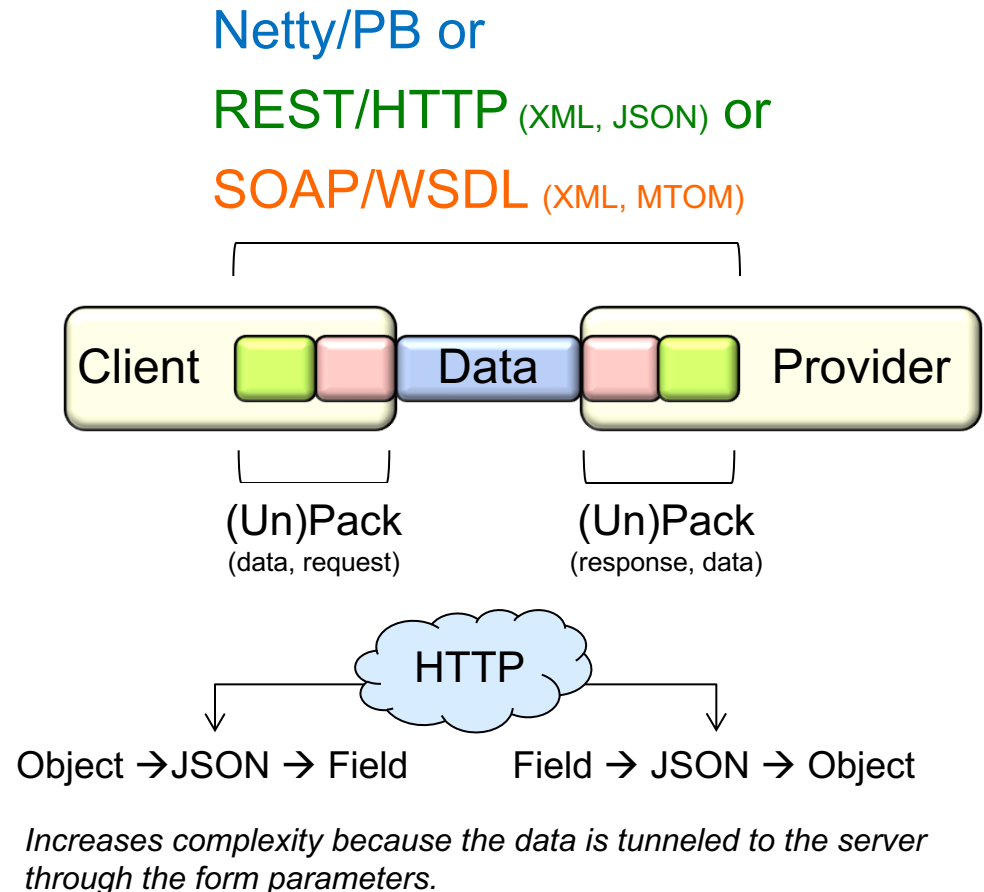
# Transmitting complex data: Overloading

- How to represent data (pass between client and server, components)?
  - ♦ Attachments (e.g., Multi-part)
    - Data is stored as a separate document attached to the request (upload, download a file) or referenced URL (common storage)
  - ♦ Name-value pairs (attributes, parameters), tuples
    - Form-like (e.g, firstname-value, lastname-value)
    - Converting to name-value pairs is difficult if the data is complex and/or deeply nested – representing graphs of data (hierarchical), the data is flattened (row-column)
    - How to support inheritance? Data changes (releases)?

# Data payloads Representations (Netty, REST,...)

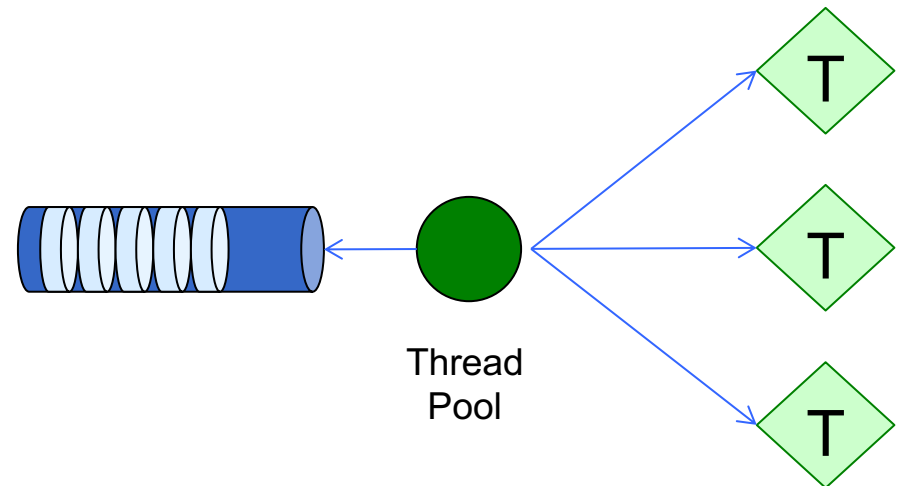
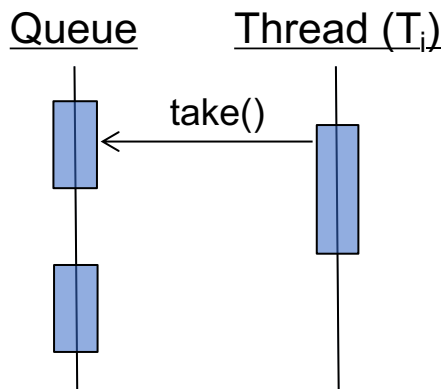
- Delivering complex data structures using overloading

- ◆ Type overloading
  - application/json, application/xml
  - Encoding data for the client and/or server to process
- ◆ Value overloading
  - Overloading POST or PUT



# The 2 diagrams used in the previous slides show different designs for delegation

- What are they?
- What are the consequences of each?



# Building an asynchronous, adhoc communication network

- Netty is a communication package built upon Java NIO
  - ◆ Focus is communication (easy to use and build capabilities)
  - ◆ Versions:
    - 3.x (sample code), **4.x (latest)**, 5.x (future)
  - ◆ Event-driven communication
    - Communication payloads are passed through a pipeline to convert from communication protocol to application data
    - Events (payloads) are
  - ◆ Asynchronous (`NioServerSocketChannelFactory`)
  - ◆ Synchronous (`OioServerSocketChannelFactory`)



# Decoding/Encoding pipeline

- Netty utilizes a pipeline encoding/decoding concept where message are passed along a stack to either retrieve from a channel (socket) or prepare to write.
- Memory pressure
  - ♦ A concern with having data pass through a Object–Binary mapping is the load it places against memory and the latency it incurs from creation and GC.
  - ♦ What strategies can be brought to the design to minimize latency?
  - ♦ Take a look at the Disruptor pattern/code

# Pipeline example

```
ChannelPipeline pipeline = Channels.pipeline();

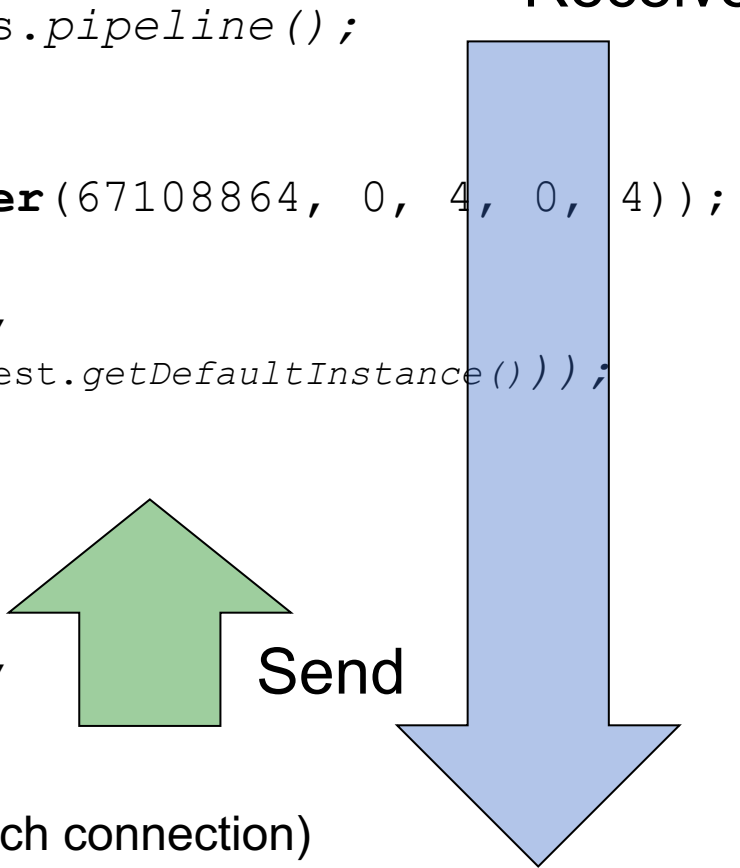
pipeline.addLast("frameDecoder",
    new LengthFieldBasedFrameDecoder(67108864, 0, 4, 0, 4));

pipeline.addLast("protobufDecoder",
    new ProtobufDecoder(eye.Comm.Request.getDefaultInstance()));

pipeline.addLast("frameEncoder",
    new LengthFieldPrepender(4));

pipeline.addLast("protobufEncoder",
    new ProtobufEncoder());

// our message processor (new instance for each connection)
pipeline.addLast("handler", new ServerHandler());
```



The diagram illustrates the data flow directions for the pipeline. A green arrow points upwards, labeled "Send", indicating the direction of outgoing data. A blue arrow points downwards, labeled "Receive", indicating the direction of incoming data.

# Design and building a scalable, low latency Netty server

- Design
  - ◆ Construct a package to receive messages from a Netty connection and delegate requests to appropriate classes
- Objective
  - ◆ Communication package for use by server processes that isolates (decouples) domain and business entities
  - ◆ Scalable Internal – How to manage resources within the server (process)?
  - ◆ Scalable External – How to cooperate with other processes to share work load?

# What are the design goals of the Netty-based project?

- Technical and package isolation (Encapsulation)
  - ◆ Of technologies
  - ◆ Patterns in the core-netty project
- Fix the bug in the software
  - ◆ What pattern is used?
  - ◆ Determine the sustainability of the classes
- Determine the behavior
  - ◆ On success?
  - ◆ On failure?

# Where to go from here

- The examples we worked on represent a basic communication package. How can this framework be enhanced to provide more robust behavior?
- Planning for a stronger implementation
  - ♦ Poison messages
  - ♦ Correlation IDs
  - ♦ Netty synchronous connection
  - ♦ Saturation of the socket connection
  - ♦ Managing connection failure
  - ♦ Load balancing
    - Options: re-routing, bidding, coordinator, ?

# Reading and references

- Netty (communication layer)
  - ♦ <https://netty.io> (standalone project, no longer associated with JBoss)
  - ♦ <https://developers.google.com/protocol-buffers/docs/javatutorial>
- Protobuf (data representation)
  - ♦ <http://www.codeproject.com/Articles/642677/Protobuf-net-the-unofficial-manual>
- Alternate data representation choices...
  - ♦ Cap'n Proto
  - ♦ MessagePack
  - ♦ XML, JSON, Apache Thrift, BSON/UBJSON
  - ♦ <https://google.github.io/flatbuffers/>
  - ♦ Avro

# More reading

- Disruptor – High performance queuing
  - ♦ <http://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf>
- Other
  - ♦ <http://dev.hubspot.com/blog/bid/64543/Building-a-Robust-System-Using-the-Circuit-Breaker-Pattern>
  - ♦ <http://www.cs.wustl.edu/~schmidt/PDF/proactor.pdf>
  - ♦ Release It!, Michael Nygard, 2009,  
<http://pragprog.com/book/mnee/release-it>
    - (check out your favorite online book store for options)
- DES
  - ♦ <http://heather.cs.ucdavis.edu/~matloff/156/PLN/DESimIntro.pdf>

## Backup slides



**We can combine these patterns to help  
create a robust server architecture**

- Patterns?
- Benefits?
- Drawbacks?

a.k.a. “The Queue”

