CmpE 275

Section: Everything we need to make the Kitchen Sink

What is message passing (to you)?

Decomposition of data

(streaming, failure detection, and patterns)

V 2.01 (discussions)

Yes, everything is an event…

Agenda

- Lecture
  - Messaging 202.1
  - 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

Lab

1. Setup network
2. gRPC/Protobuf

Lab work for this week

(Note blocking and queue subparts)

- Basic: Explore grpc and the <u>blocking</u> code base
  - Create a chain of three processes (A, B, C)
  - Circuit breaker pattern between nodes B and C
- Strive to
  - Heartbeat between nodes
    - Can you make the algo more aggressive, less aggressive?
  - Scale message traffic – How would you add QoS behavior and where (A, B, or C)?
- Advanced: Ordered processing
  - Using the <u>queue</u> code base, implement out-of-order support. Hint: Construct a diamond pattern (ABCD)

Class Discussions

- RL1
  - **Group topics must be unique!**
    - Coordination between teams is needed? How do you do this?
    - Model this? What pattern tools are you going to use?
  - Lecture uploaded about the RLs (reverse-lecture-project-info.pdf)
  - We are not looking at application functionality/development. Rather digging deeply into the design (like slide 15).
  - **Email: Use of RabbitMQ or other MQ?**
    - ANS: No. Reason is the MQ is introducing a fourth server (the MQ is process) therefore the overlay (graph) resembles a spoke-hub (MQ).
    - E.g., not the overlay we are looking for:
      - A-MQ-B, B-MQ-C, etc
  - **Slack: Can we use sockets, or gRPC, or (??) ?**
    - Yes, though not anything that adds another process (e.g., MQ). E.g., Netty.
    - Stay away from language only implementations
  - **Talked about: SPOF, Memory pressure, Motion as topics (slide 14)**
  - **You can use code from the labs for RL1**
  - **What should be in the report?**
    - **TODO: I will put together a pdf with goals, limits, and report format/content**

Class Discussions (pg 2)

- Labs
  - Lab 01: basic components of client-server (comm/sockets message/text)
  - Lab 02: Note uploaded **lab 02 (help)** – provides a tool to download the gRPC jars, and keep them in one directory
    - Flatten the directories for gRPC dependencies.
    - Include in your project (aka eclipse IDE, or ?)
    - Spend some minutes looking the lab (blocking)

Let’s look at reading a file (transporting data from disk to memory)

- Send (write) and receive (**read**) are like:

```
FileInputStream fis = null;
try {
   fis = new FileInputStream(fn);
   final int blen = 1024;
   var raw = new byte[blen];
   var done = false;
   while (!done) {
       var n = fis.read(raw, 0, blen);
       if (n <= 0) break;
   }
} catch…finally { fis.close(); }
```

Steaming is a similar concept of reading and writing to file…

Streaming or event processing is the decomposition of a larger conversation into many (N) parts. It is also referred to as chunking, paging.

- As in data, e.g, reading a file (byte[])
- Or for a process (actions) into events

Examples: search engine results (pages), database result sets, tiling, links

This decomposition is also referred to as Discrete event modeling

- DES (Discrete Event Modeling) is used to create simulations of complex systems to understand/predict behavior given a set of initial assumptions/conditions
  - Uses
    - Load/Stress testing
    - Prediction: Modeling, Financial systems
  - Can we utilize DES concepts in the application of distributed systems?

Types of DES

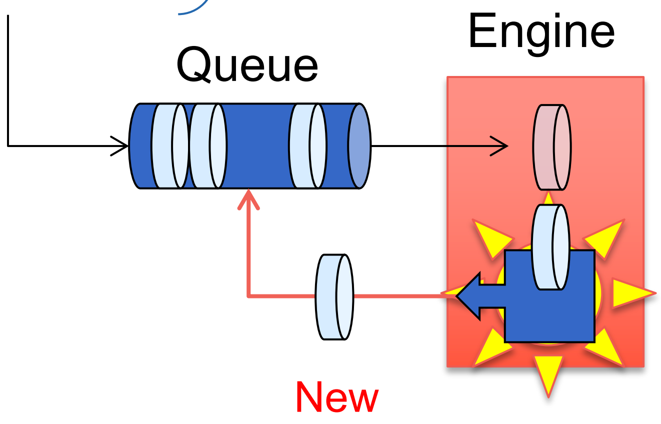
- Conservative
  - As events (time) is irreversible. Once execution occurs, it cannot be unwound if new events occur after the current simulation time (e.simTime < engine.simTime)

- 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+Engine

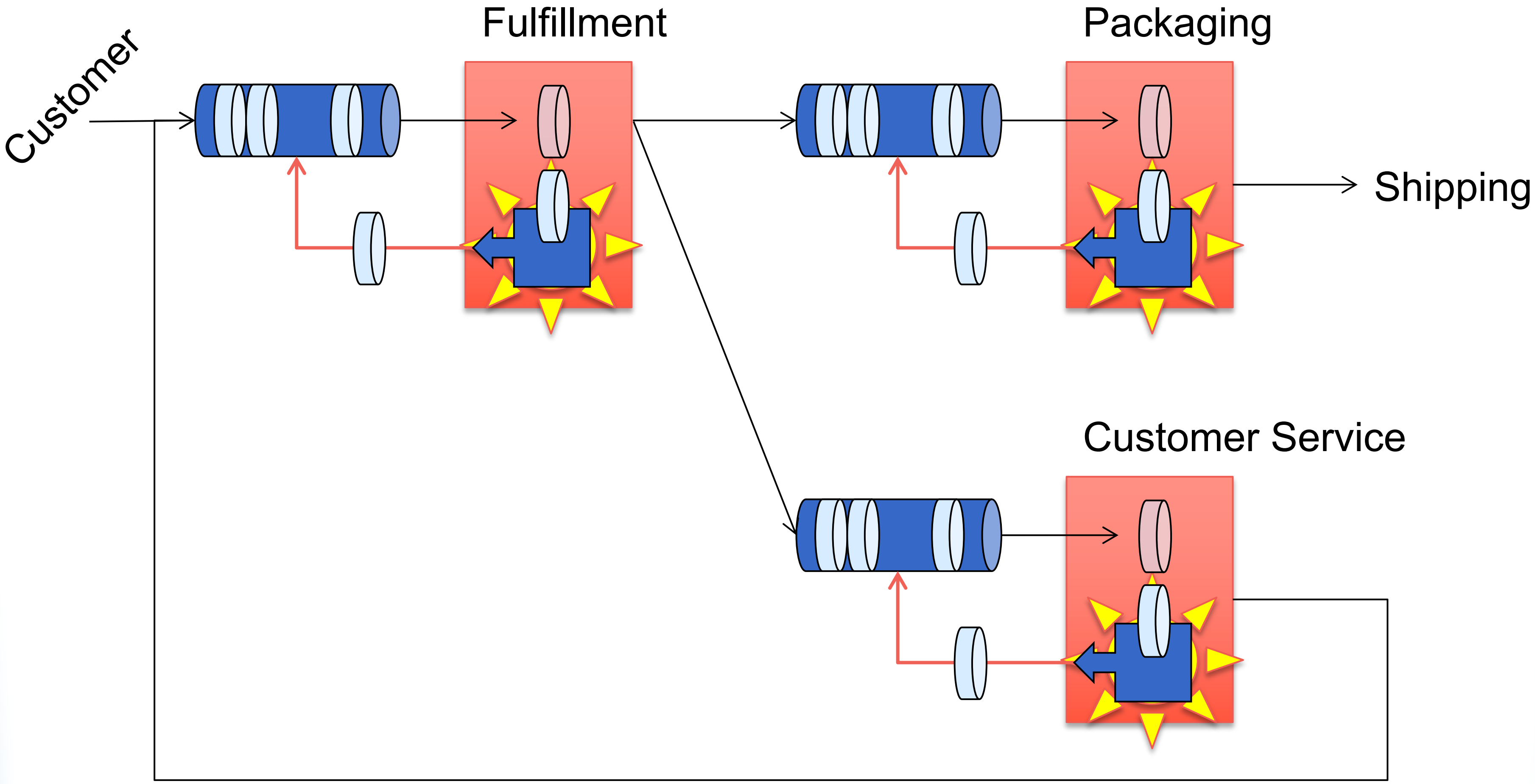
- The event engine is a queue that is sorted on simulation time
  - Time is continuous (float) s.t. an one can always insert an event between two events (given e1.time != e2.time)

```
Event {
   simTime
   data|action
}
```



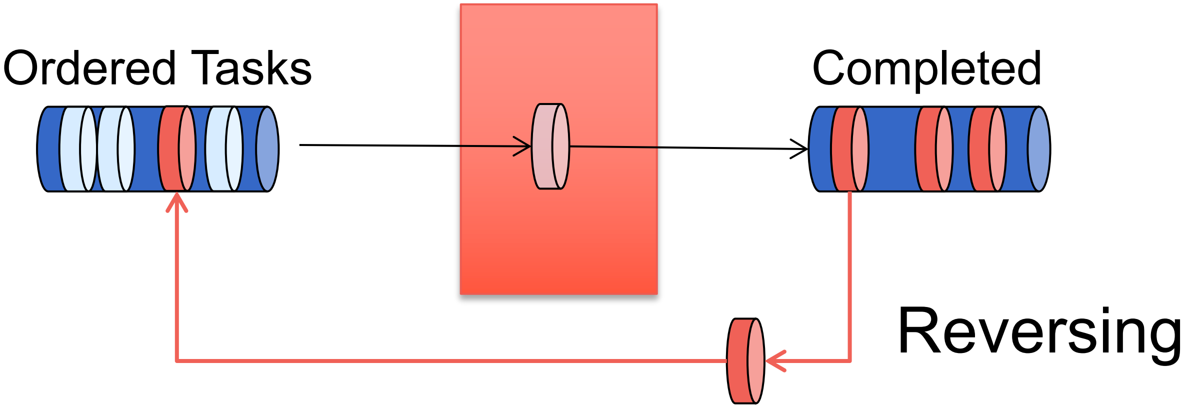
```
while (eventsExist) {
    list = eventsAt(simTime);
    foreach e in list
        execute(e)
    simTime += dt;
}
```

Complex systems are a set of discreet steps (multiple Queue+Engine)



DES: Reversing simulation time

- Asynchronous events may arrive to a server after the servers simulation time.
  - Options
    - Ignore the late 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 (enqued elements can be nondeterministic)
- 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();
```

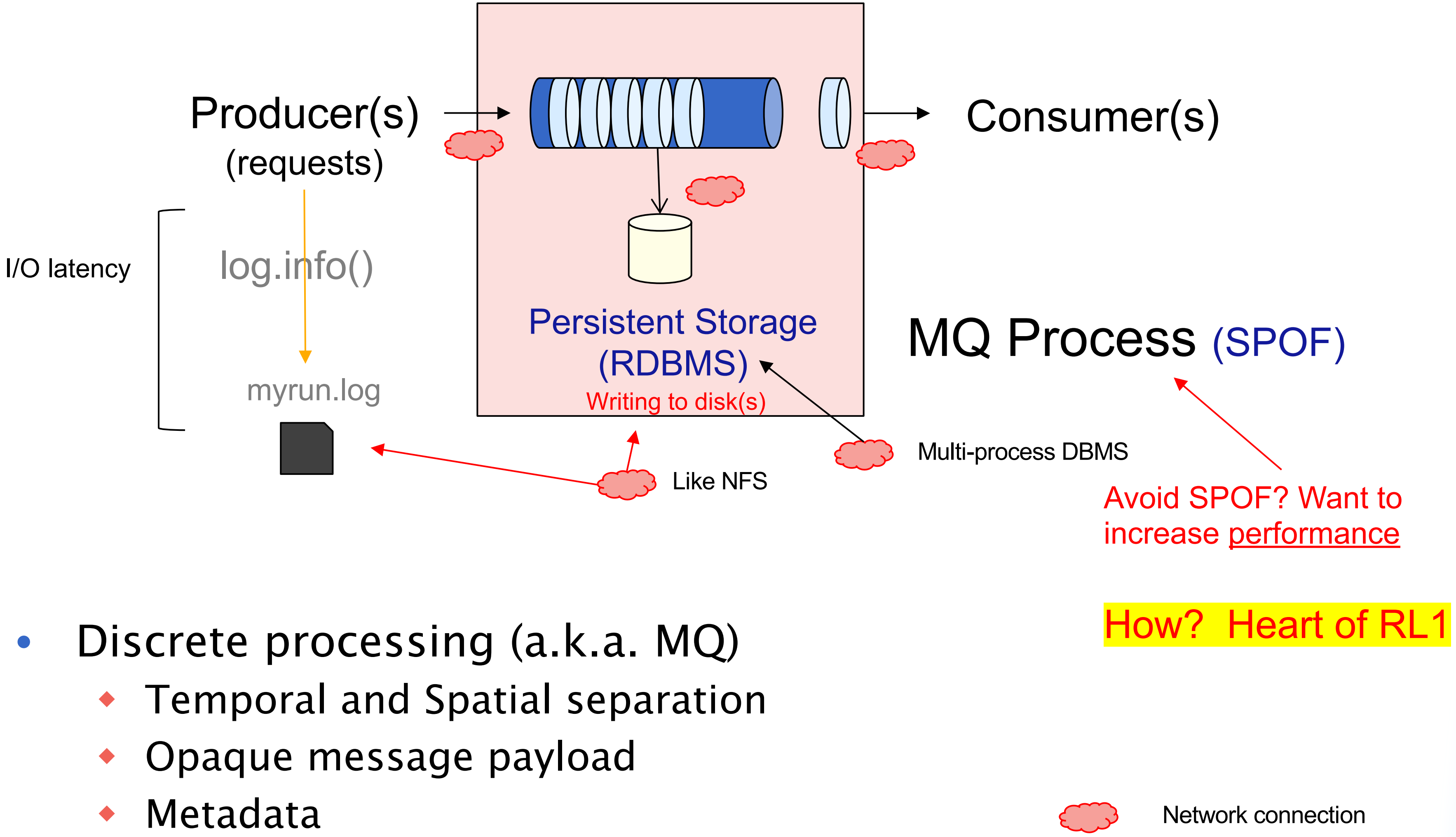
How do we apply DES to getData() – could be a RL1 topic (refer to next slide for example)

The are:

1. Reversing (error recovery), minor: decomposition strategies
2. Memory constraints (queue, in flight data)
3. Roles/Decomposition strategies
4. Expand to cover SPOF removal/limiting?
    1. Require modification of our overlay network (may need to extend ABC)
5. How to modify ABC to avoid SPOF (not at a functional level, rather looking at latency, pressure, and motion?)
6. ?

Let’s see how DES applies to MQs

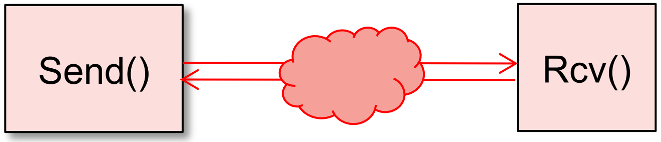
(Example of DES decomposition of a MQ and its network latencies)



How? Heart of RL1

- 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

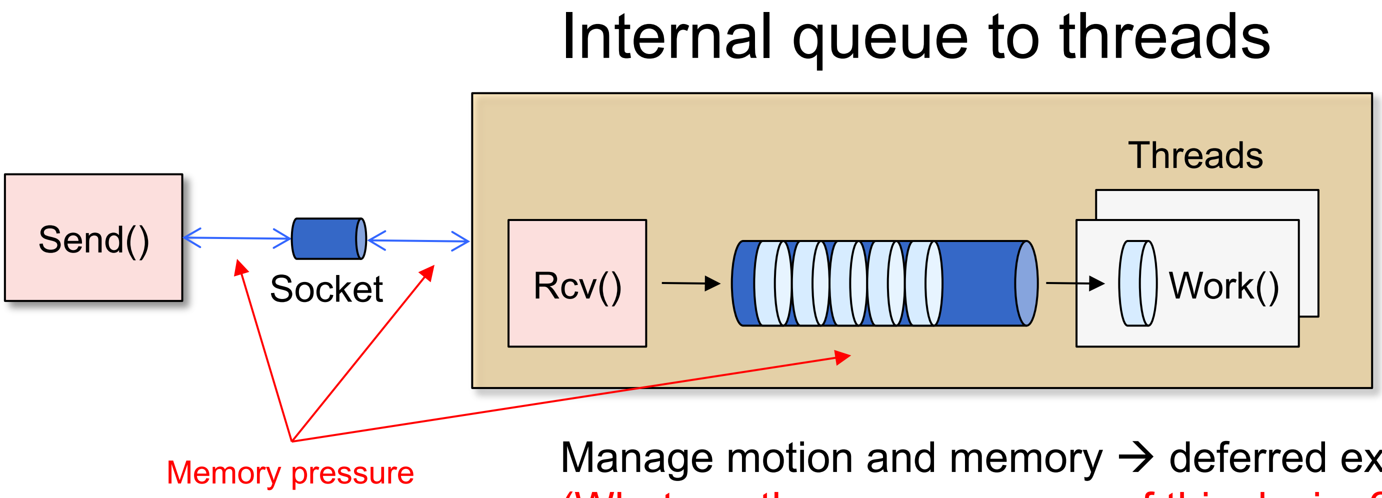


```
String getName() {
    return “foo”;
}
```

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

- Synchronous?
- Asynchronous?

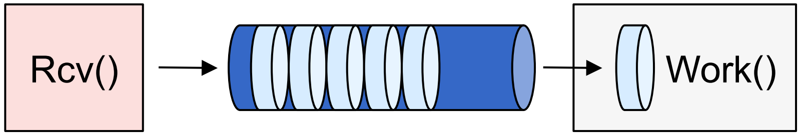
MQ [DES] design applied to load balancing



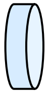
Manage motion and memory → deferred execution
(What are the consequences of this design?)

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

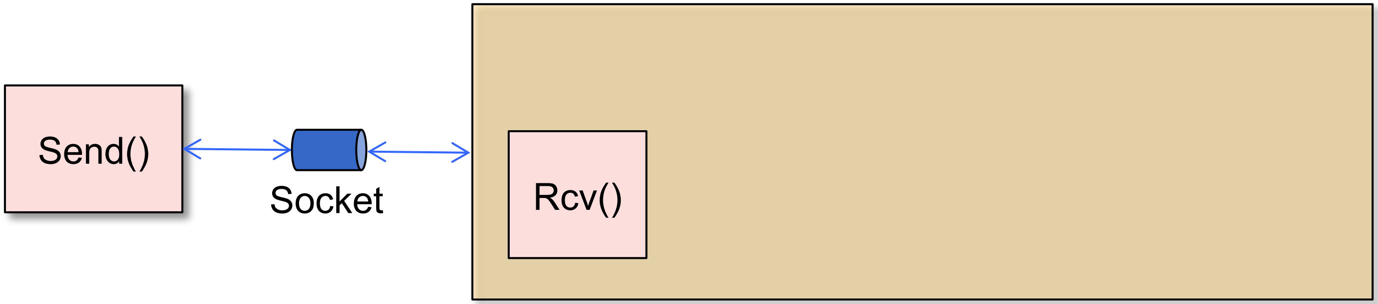
Looking at the internal relationships



- What is the behavior between Rcv() and Work()?

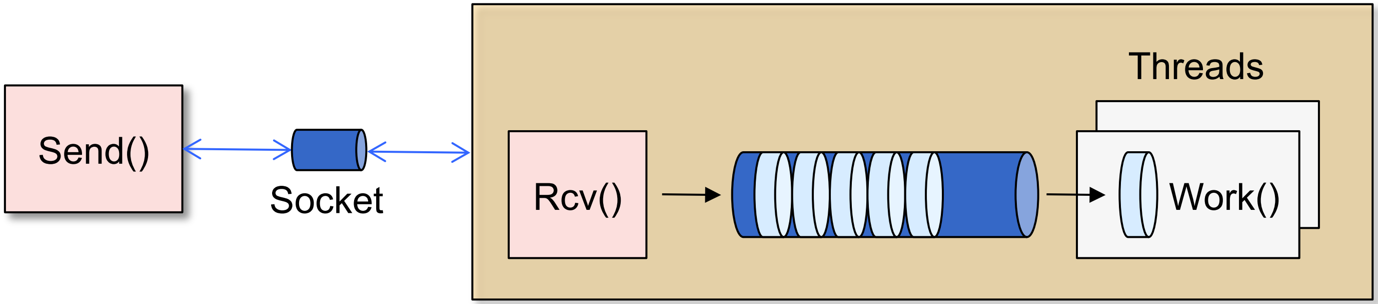
- What are the possible strategies when these components are under stress ( >>)?

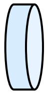
Looking at the external connection



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

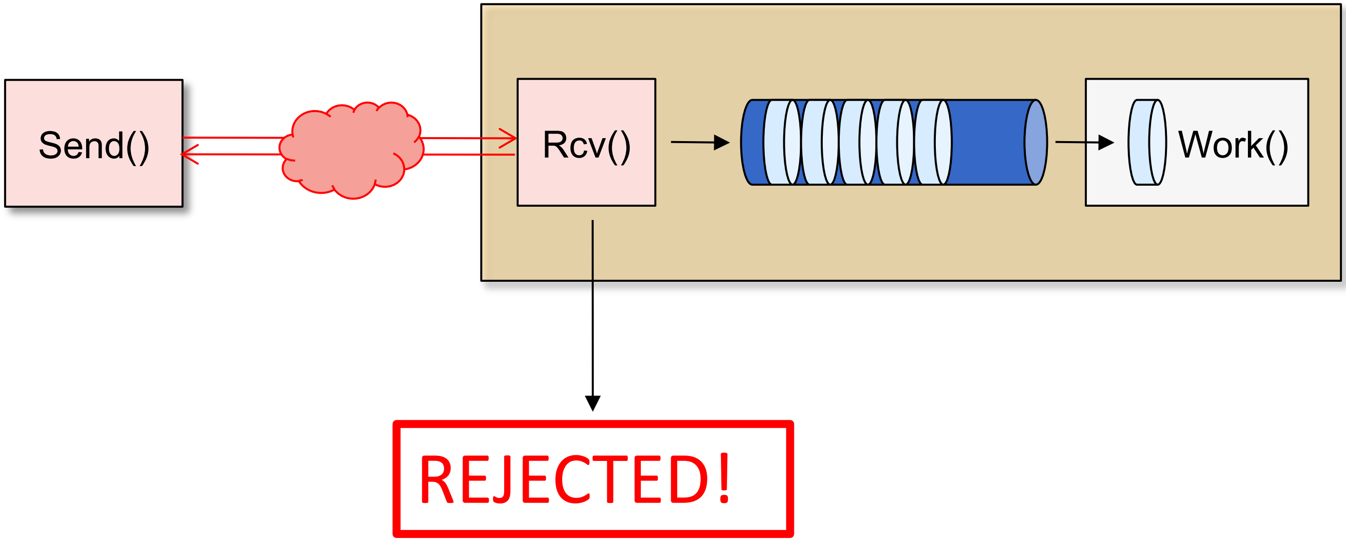
Applying DES to our work
(managing stress)



- 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

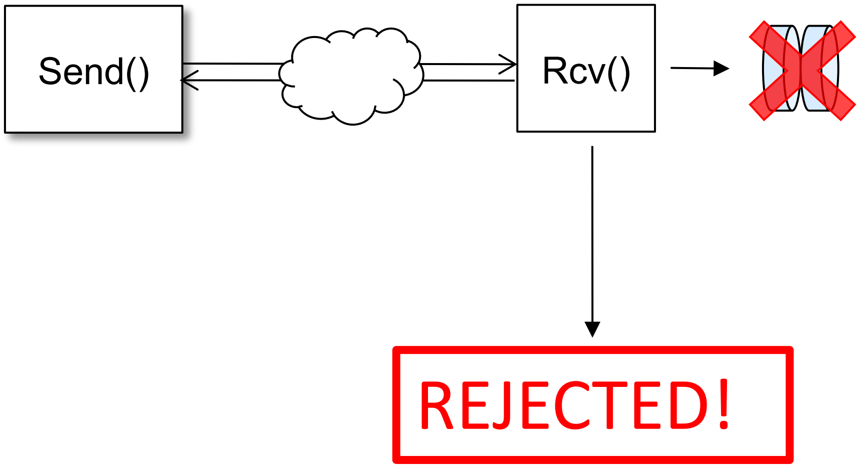
Queues vs. Fail fast

- 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?

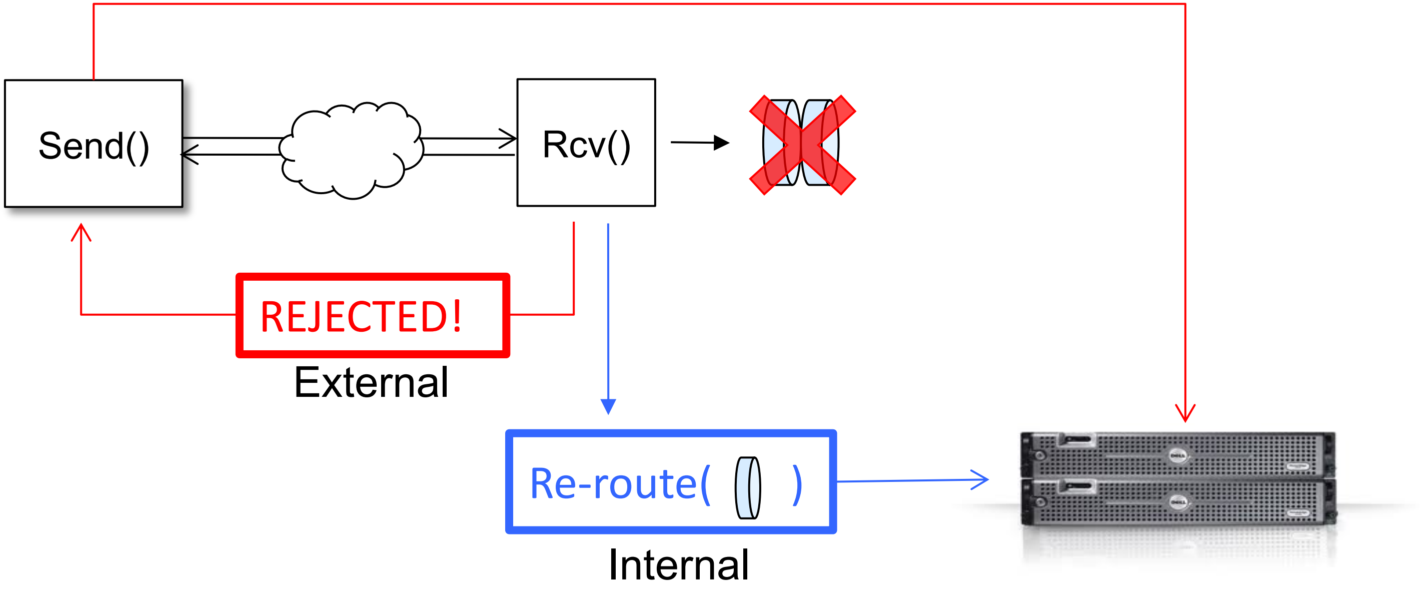


- Failures will occur (Finagles' Law)
- Writing perfect code is hard ($$)
- A recovery strategy can complement scaling strategies

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)

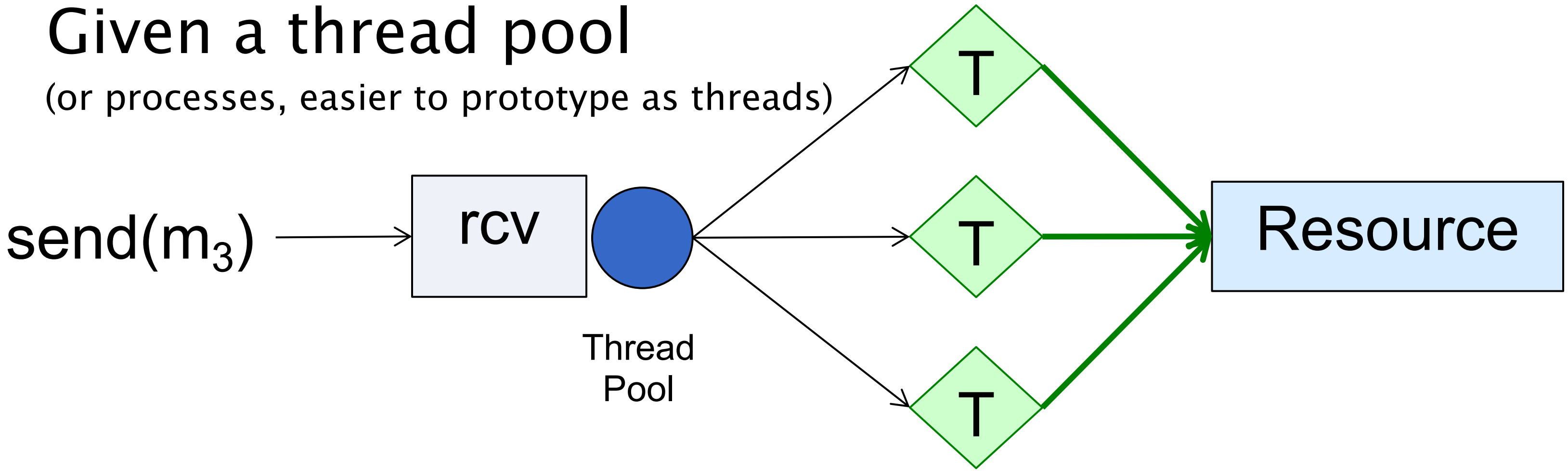


Pattern: Circuit Breaker

- 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.

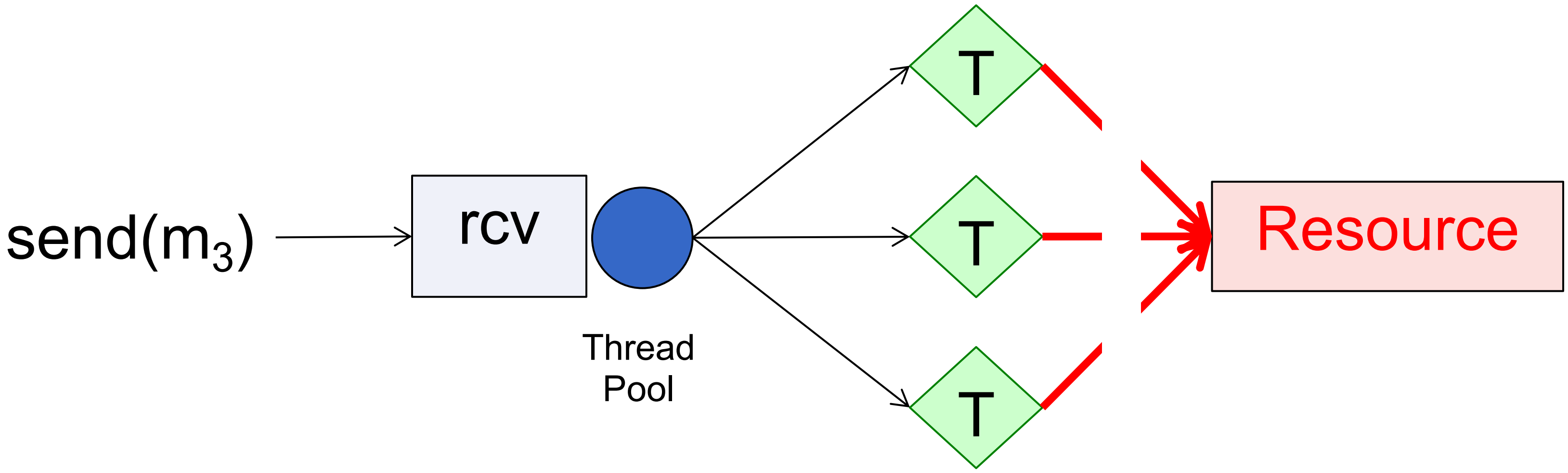
- Given a thread pool

  (or processes, easier to prototype as threads)



On failure how do each of the threads (T) or processes respond?

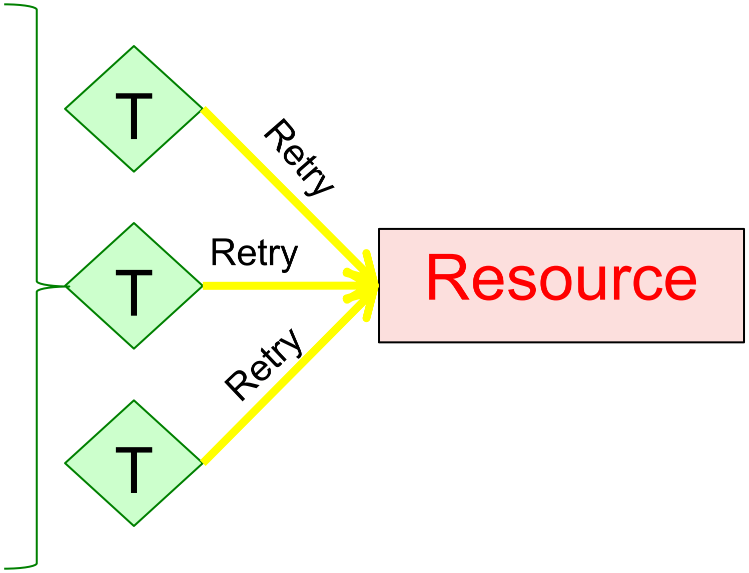
- Let’s assume that while the system is running, the resource (or the connection to) fails.
  - What is the perspective of each thread?
  - Remember: Isolation of each thread is desired



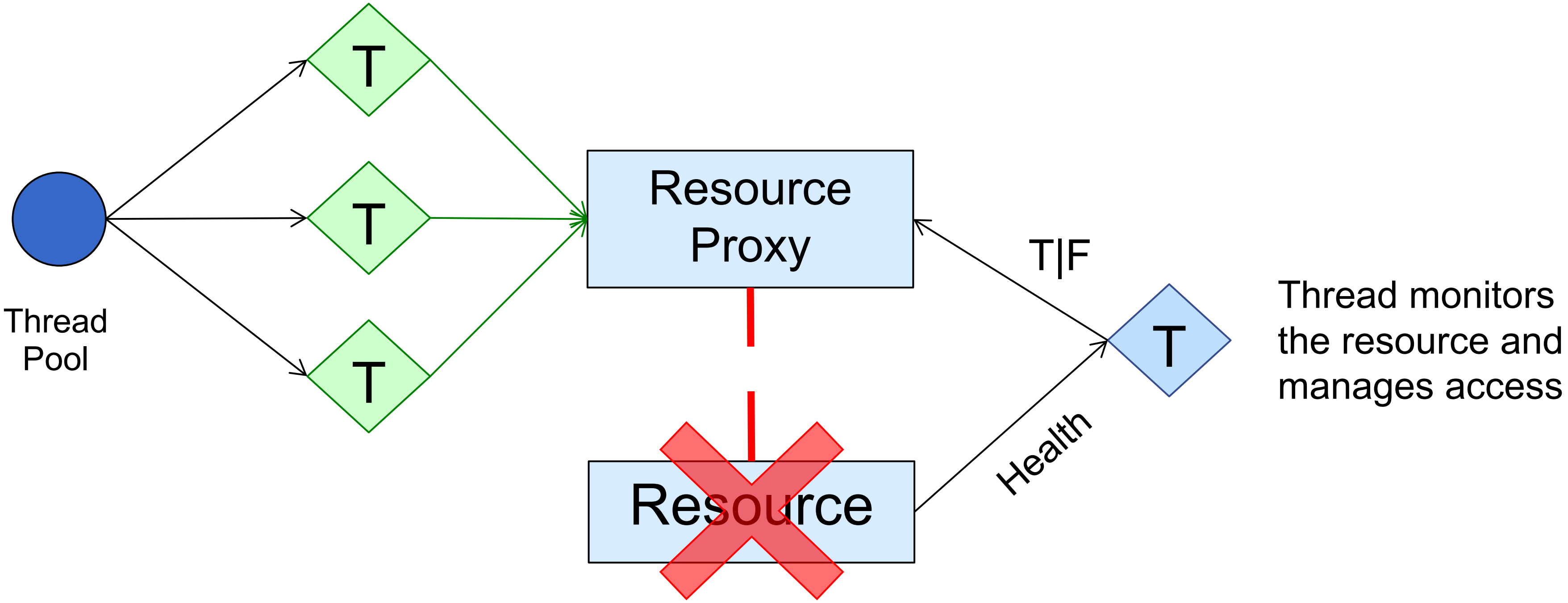
CB: uncoordinated access to a resource can consume resource cycles, propagate error messages and impact overall performance w/ failure-retry logic

- 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 and be expensive (e.g., DB connection, sockets)

```
for(int n = 0 ; n < N; n++) {
   if (connected)
     break;
   else {
     logger.error(“failed…”);
     DBConnection.connect();
   }
}
```

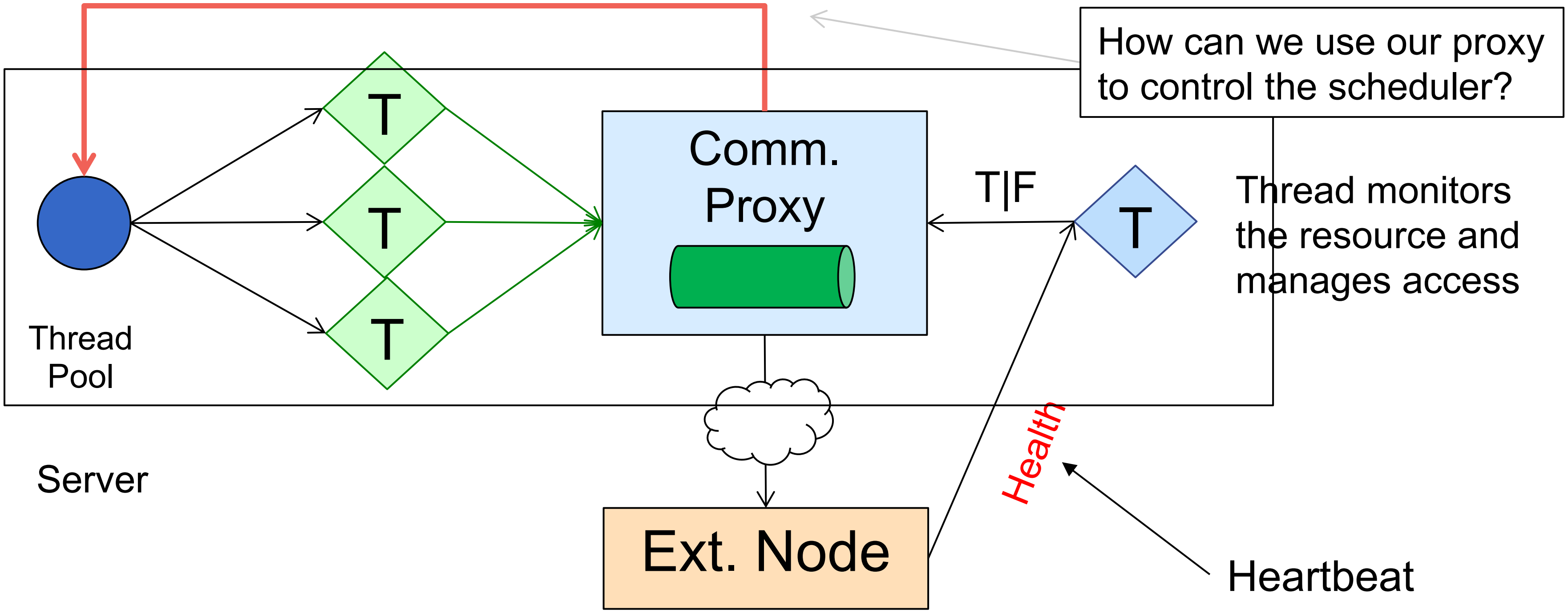


CB pattern is employed to help reduce the retry and message propagation



- 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)

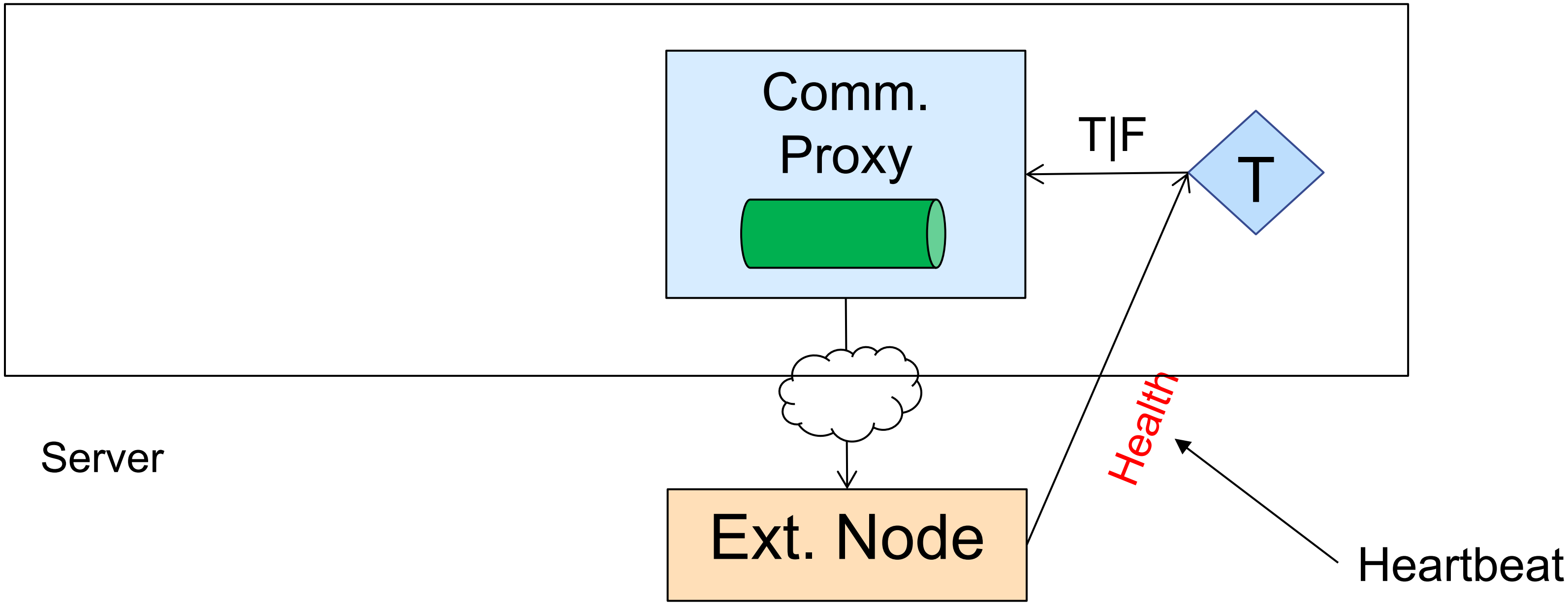


How can we use our proxy to control the scheduler?

Thread monitors the resource and manages access

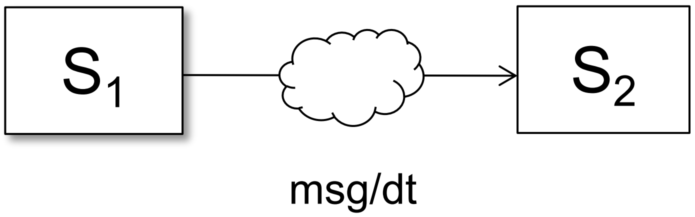
- We have applied several patterns here to create a robust resource manager
- **Patterns: Proxy, Queue, Heartbeat, Circuit Breaker**

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

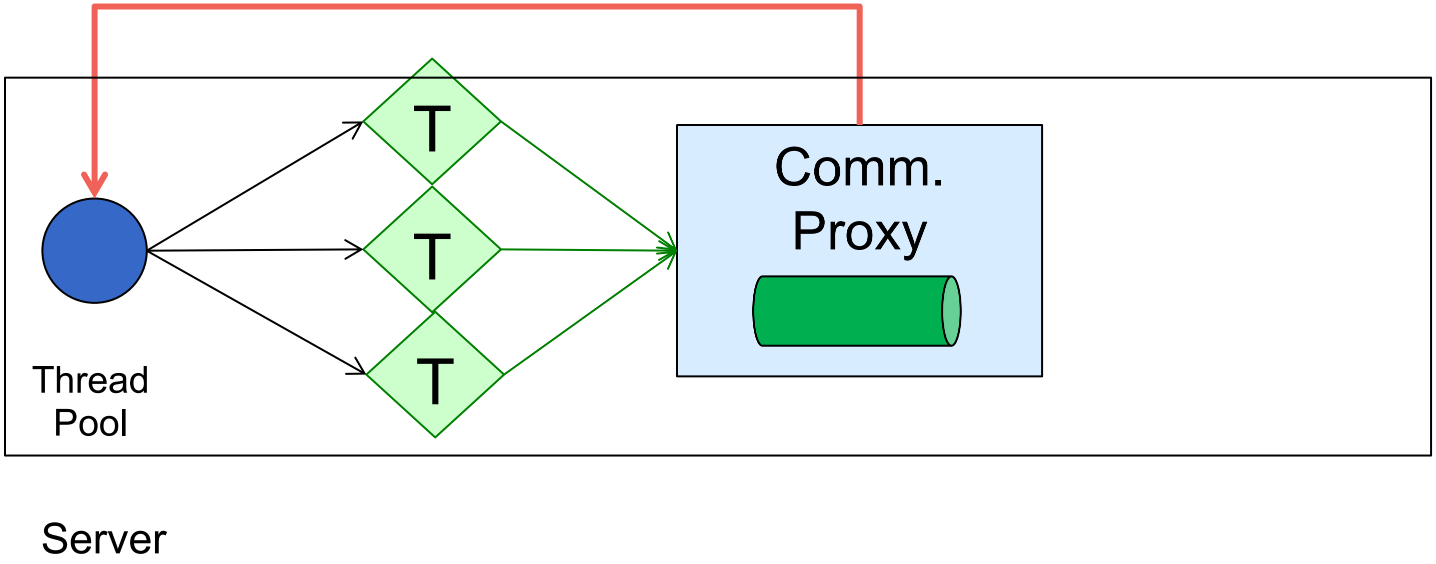


- One-way message
- Lightweight
  - Small memory overhead
  - Frequency doesn’t impact network
- Detection
  - Missing sequential messages could indicate a problem

```
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



- 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?**

Other Patterns: Bulkhead

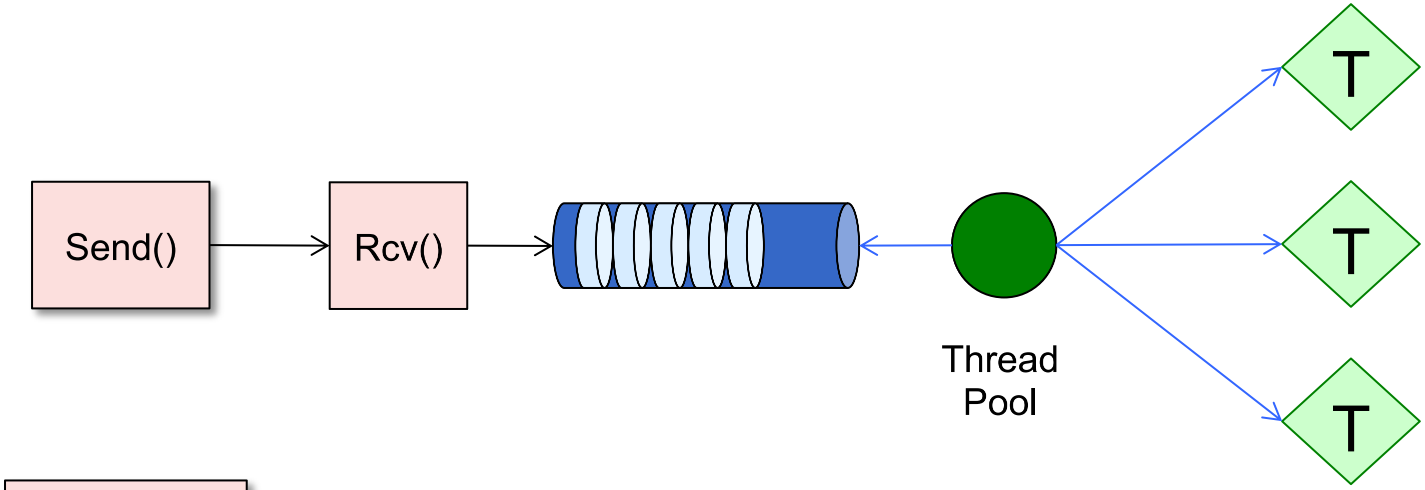
- Bulkhead protects the system against resource failures causing cascading problems to the system
  - hardware isolation and redundancy
  - Thread management – thread pools
    - Isolate/Reserve threads for management/monitoring use vs. runtime use – this ensures that if a problem occurs with the public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

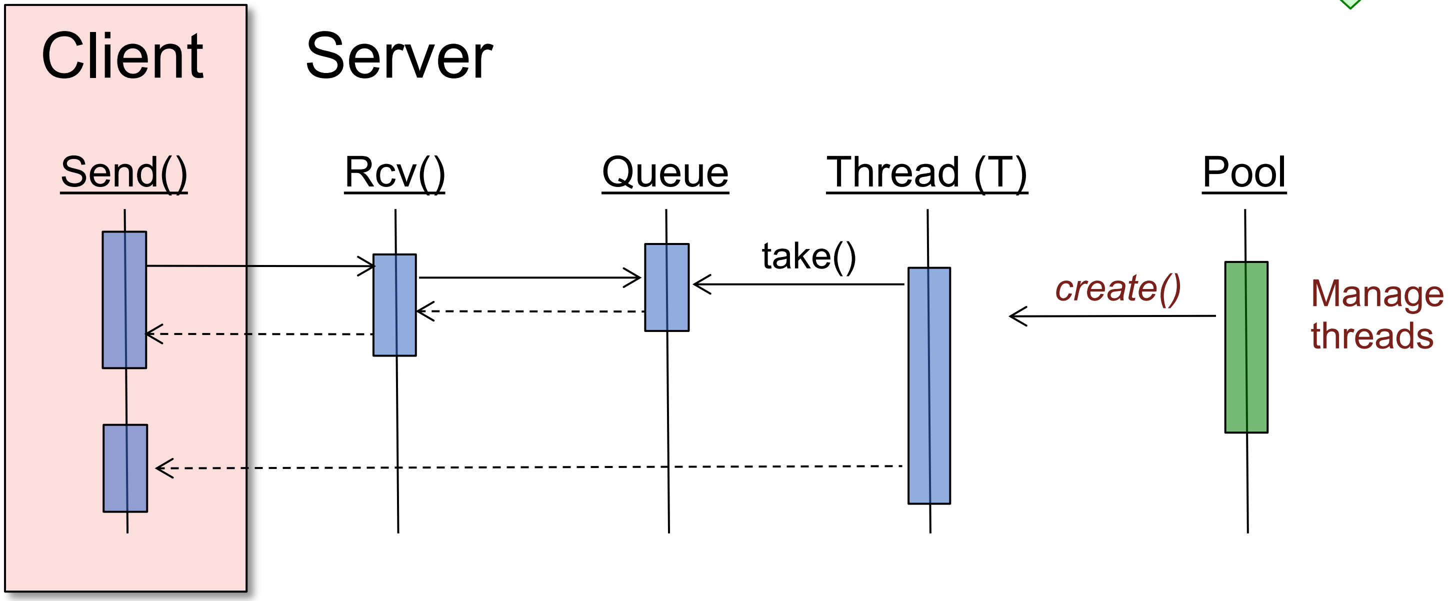
Pattern: Proactor

- Asynchronous event handling
  - Queues
  - Thread pools

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

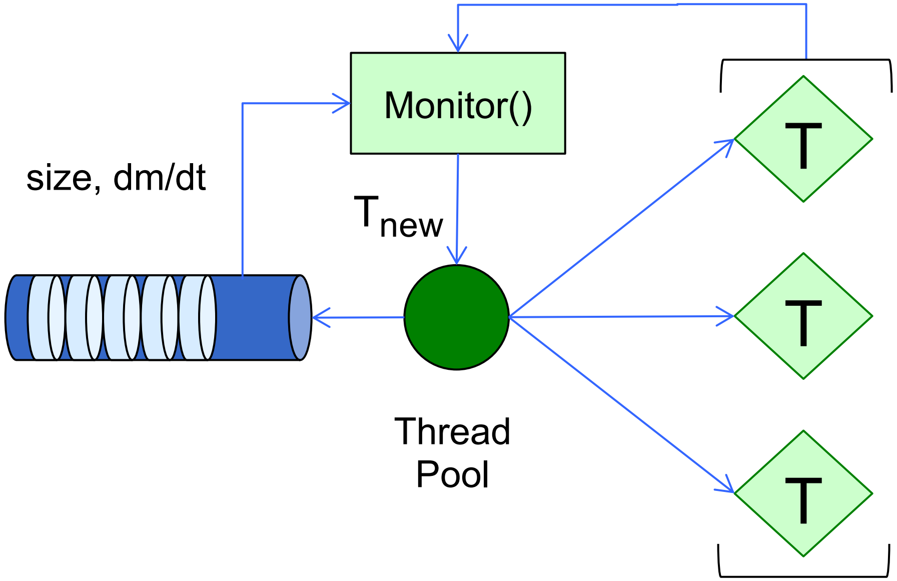
Proactor combines queues and thread pools to provide asynchronous message processing/scaling

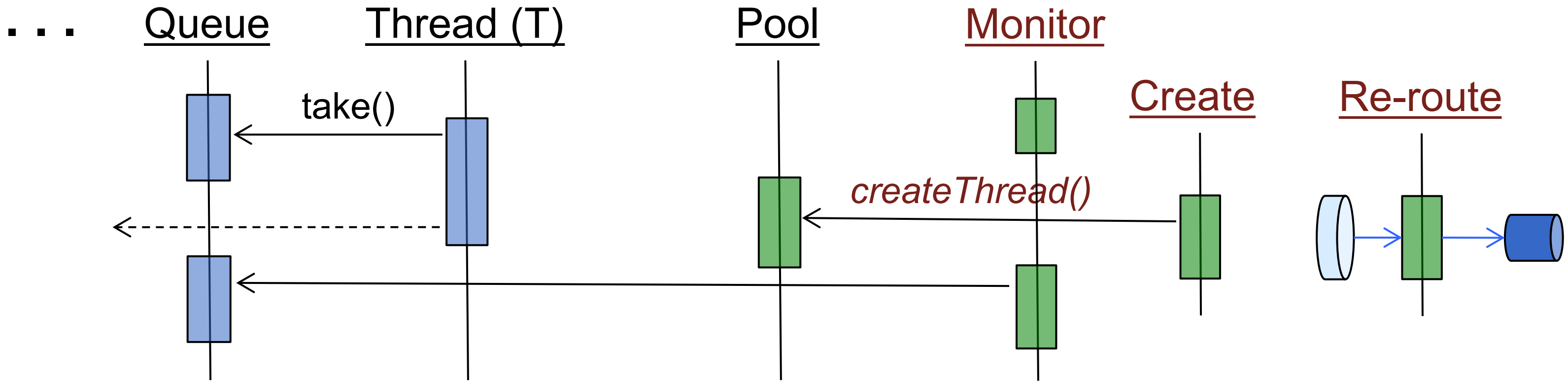




A Proactor design with feedback for adaptive queue management

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





Revisiting the asynchronous get Data

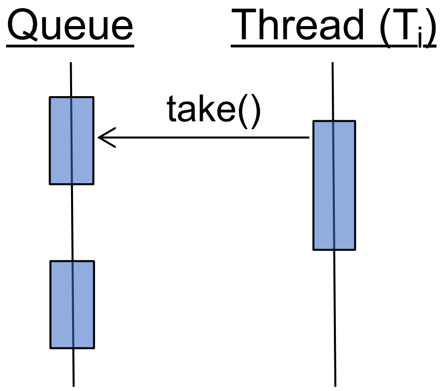
```
Data getData();
```

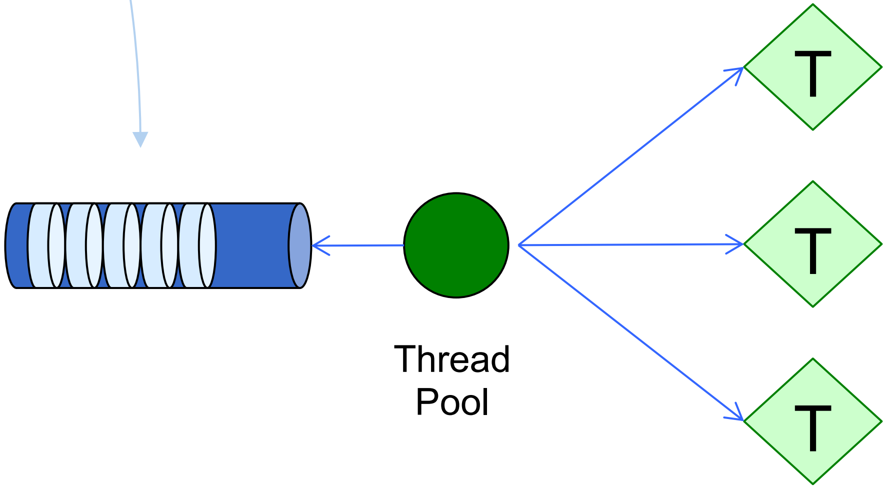
```
Data d = getData();
If ( d != null ) process(d);
```

What if we decomposed the problem into smaller steps?

What are the consequences of delegation?

- Single vs. Multiple threads?
- FIFO-based designs?





Appendix: Netty Coding Explained

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

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:
    - **4.x (lab)**, 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 `(NioSeverSocketChannelFactory)`
  - Synchronous `(OioServerSocketChannelFactory)`

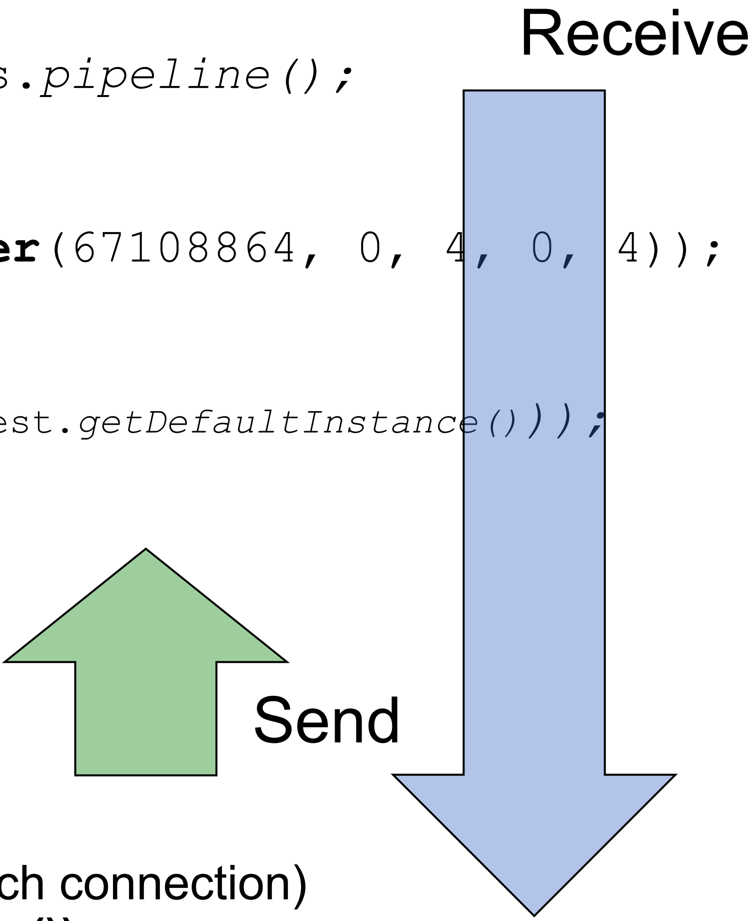
Decoding/Encoding pipeline

- 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());
```



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
  - 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, BJSON/UBJSON
  - https://google.github.io/flatbuffers/
  - Avro

More reading

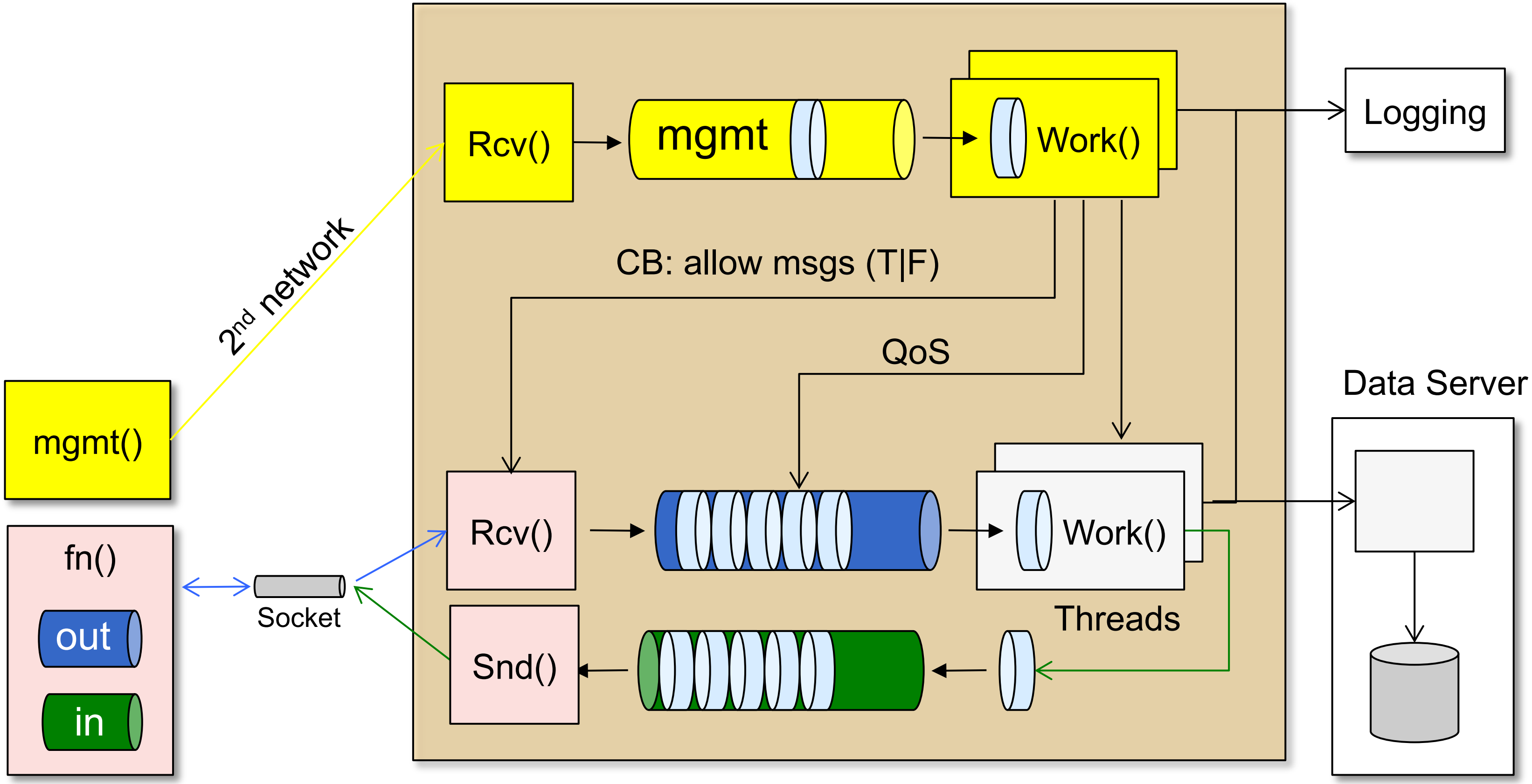
- Disruptor – High performance queuing
  - http://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf
- Other
  - http://dev.hubspot.com/blog/bid/64543/Building-a-Robust-System-Using-the-Circuit-Breaker-Pattern
  - http://www.cs.wustl.edu/~schmidt/PDF/proactor.pdf
- DES
  - http://heather.cs.ucdavis.edu/~matloff/156/PLN/DESimIntro.pdf

Backup slides

Combining concepts to help create a robust server architecture

- Identify Patterns?
- Benefits?
- Drawbacks?

Sever Internals



Rcv()
mgmt
Work()
Logging
CB: allow msgs (T|F)
2nd network
QoS
Data Server
mgmt()
Rcv()
Work()
fn()
out
in
Socket
Threads
Snd()