AARHUS UNIVERSITET

Microservices and DevOps

Scalable Microservices

Stability Patterns

Henrik Bærbak Christensen

Anti --- Antipatterns

- Nygaard lists 14 antipatterns
  - Users, blocked threads, slow response, chain reactions, etc.

- So – how to combat these???
  - The *Stability Patterns*
  - (and the ‘Remember this’ section in each antipattern chapter, which mentions stability patterns that are not mentioned in this chapter, hmmm)

Summary

- Note: Figure from *first edition!*
- Missing in 2nd Ed, and a bit different pattern set
- But – nevertheless…

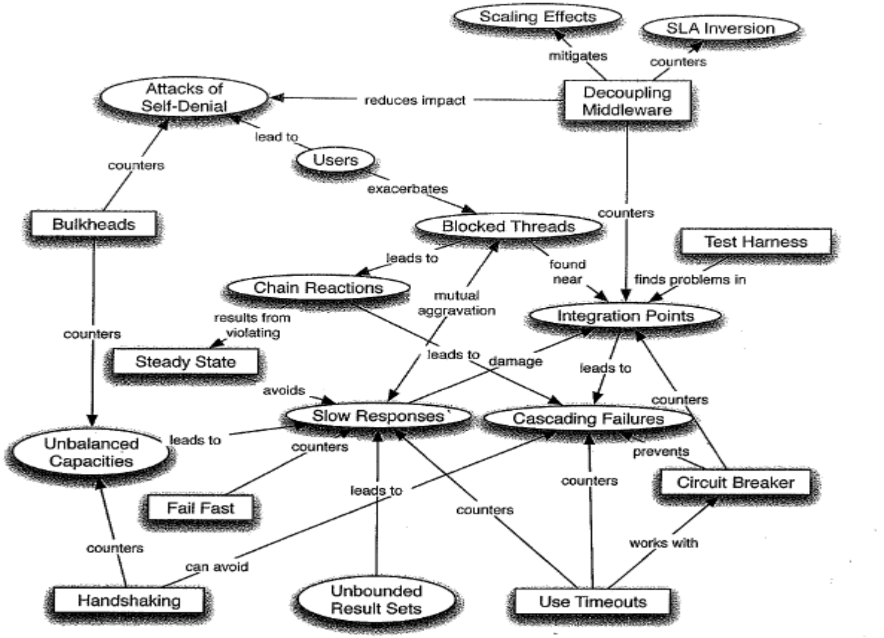


Figure 3.1: INTERACTION OF PATTERNS AND ANTIPATTERNS

Is All This Clutter Necessary?

- As stated by Nygard:

- *… handling all the possible timeouts creates undue complexity in your code. It certainly adds complexity.*

- *… Your users may not thank you for it, because nobody notices when a system* ***doesn’t*** *go down, but you will sleep better at night.*

Timeouts

- *Timeouts:* Guard *any* call to remote units with timeout to avoid waiting forever on an answer that never arrives

  – Well placed time-outs provide fault isolation: your code does not break due to a failure in another subsystem.

  – Beware of vender supplied APIs, sometimes they have forgotten to add time out parameters ☹

    - Or it is in the API, but not implemented ☹ ☹ ☹

Timeouts

- *Timeouts:* Guard any call to remote units with timeout to avoid waiting forever on an answer that never arrives

  – Apply to Integration Points, Blocked Threads, and Slow Reponses

  – Consider *delayed retries*

  - Network issues take time to go away – do not retry again immediately

    – See ‘Retry’ pattern (that I introduce) later…

Timeouts on Locks

- Java supports running multiple threads
  - To avoid ‘weird stuff’ you need to guard ‘critical regions’
    - i.e. the method that two or more threads may call at the same time
- Classic Java
  - Synchronized methods cannot time out!
- Modern Java
  - ReentrantLock myLock;
    - Acquire not by ‘myLock.lock()’ but…
    - By ‘myLock.tryLock(3, TimeUnit.SECONDS)’

AARHUS UNIVERSITET

Circuit Breaker

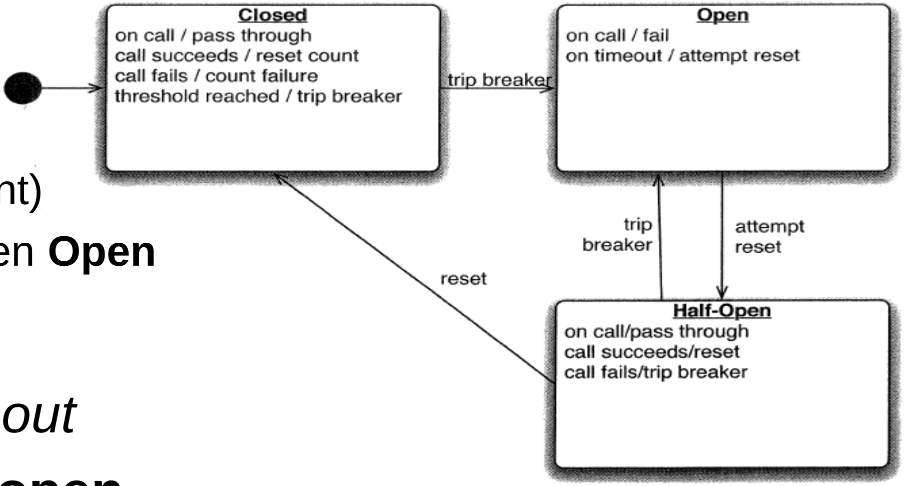
- The idea of a *fuse:* Burn before your house does!
- *Circuit breaker:* Da: Maksimalafbryder
  - *… en automatisk afbryder, som bruges til udkobling af overstrømme.*
  - *Består af bryderdel og relædel. Relædelens funktion er at måle strømmen og i tilfælde af en overstrøm udkoble af-bryderen*
  - HFI relæ

Circuit Breaker

- *Circuit Breaker:* Wrap dangerous operations with a component that can circumvent calls when the system is not healthy. Differs from retries, in that circuit breakers exist to *prevent* operations rather than to re-execute them.

Circuit Breaker

- **Closed** state:
  - Execute operation as normal
    - If operation fails (timeout) then
      - Note it (increment failure count)
      - If failure count > threshold then **Open**
- **Open** state:
  - *Fail fast, avoid the wait for timeout*
  - After a set time, switch to **Half-open**
- **Half-open** state:
  - Execute operation again
    - if fail then goto **Open** state immediately
    - if success then goto **Closed** state



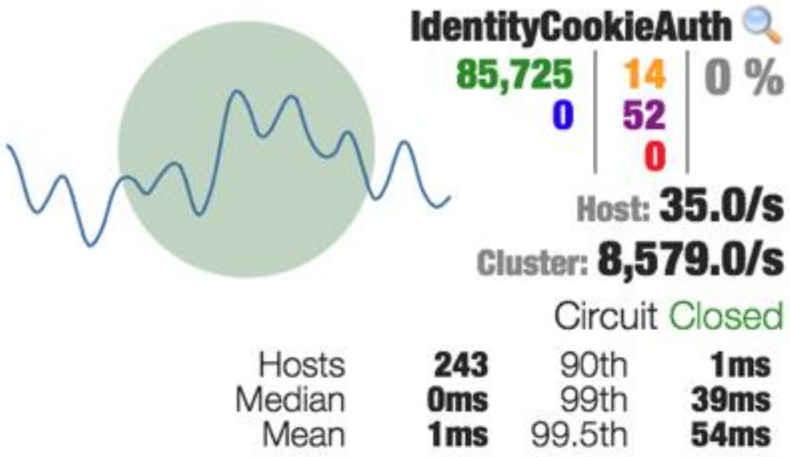
Exercise: Why the need of the half-open state?

Circuit Breaker

- Is a way to make ”graceful degradation”
  - Degrade functionality when under strain
  - Avoid DogPile, as it gives producer service time to recover
  - Important to involve stakeholders
    - ”What to do if in the open state?”
      - What to do if we cannot verify credit card?
- Remember
  - Use together with timeouts
  - Expose, track and report state changes
    - Log any ”popped fuses”!

NetFlix

- NetFlix is a large micro service system, based upon circuit breakers.
  - Open sourced their circuit breaker implementation w. dashboard



IdentityCookieAuth

85,725 | 14 | 0 %
0 | 52
0

Host: 35.0/s

Cluster: 8,579.0/s

Circuit Closed

Hosts	243	90th	1ms
Median	0ms	99th	39ms
Mean	1ms	99.5th	54ms

Resilience4J

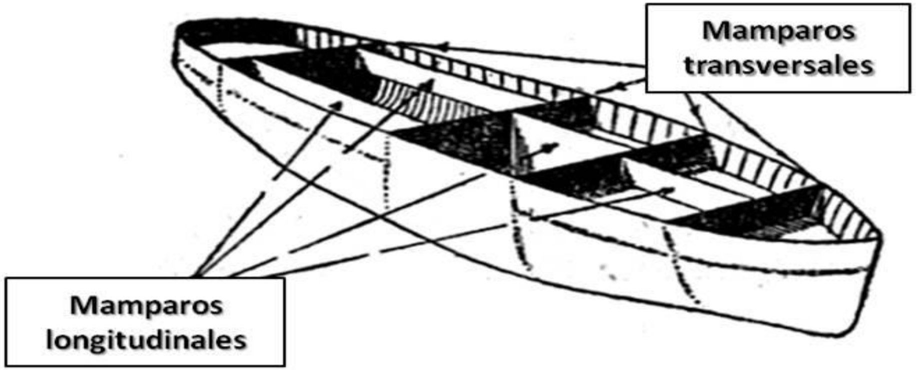
- … is
  - *A lightweight, easy-to-use fault tolerance library inspired by Hystrix, but designed for Java 8 and functional programming.*

- You can pick and choose just the piece you want
  - CircuitBreaker: Nygard’s pattern in it’s *frequency* form
  - Bulkhead: Limit number of concurrent executions
  - RateLimiter: Limit rate of requests (or queue them)
  - Retry: Retry call N times with M mS delay between
  - TimeLimiter: Nygard’s *Fail Fast* pattern
  - Cache: You guessed it ☺

AARHUS UNIVERSITET

Bulkheads

- Da: *Skot*
  - *Partitions that can be closed preventing water from moving from one section to the next*
  - *Damage containment*



- *Bulkheads:* Partitioning a system so failures in one part does not lead to system failure

- Simplest (most common) form: **Redundancy**
  - Have two or more servers handling the load

Bulkheads

- Mission-critical form:
  - Pool of servers/services reserved for critical use while the rest are available for non-critical use
- Example
  - Servers dedicated to airline check-in (critical)
  - Others serve flight status checking (non-critical)
- Liability (see next slide)
  - You now have two disjoint resource pools that are subject to *unbalanced capacities*
  - I.e. you need more reserved capacity

Bulkheads

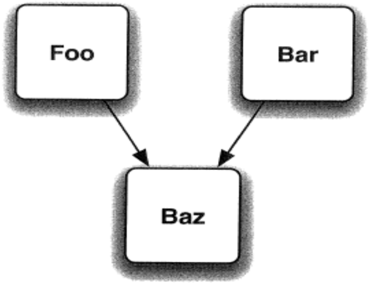


Figure 5.2: Hidden Linkages

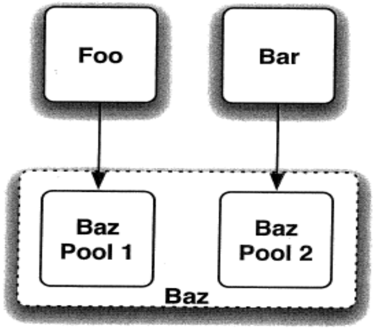
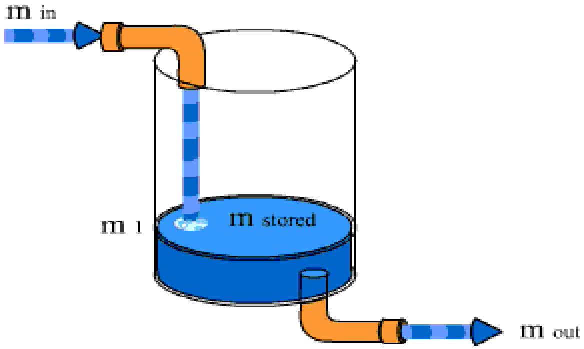


Figure 5.3: Partitioned System

Steady State

- *Steady state:* For every mechanism that accumulate resources, some other mechanism must recycle that resource.
- If not, accumulated resources outgrow capacity
  - Log files, DB rows, caching, …
- If capacity is exceeded - bad things happen



Examples

- MongoDB
  - Allocated disk in max 2GB chunks
  - If it cannot it simply stops processing write requests, only processing read requests
    - Will not reenable write requests until after restart
    - (and data purge and compacting is a write request ☺)
- Docker container logs are written to storage
  - So if you never look at them, they just grow

Steady State

- Another potential definition

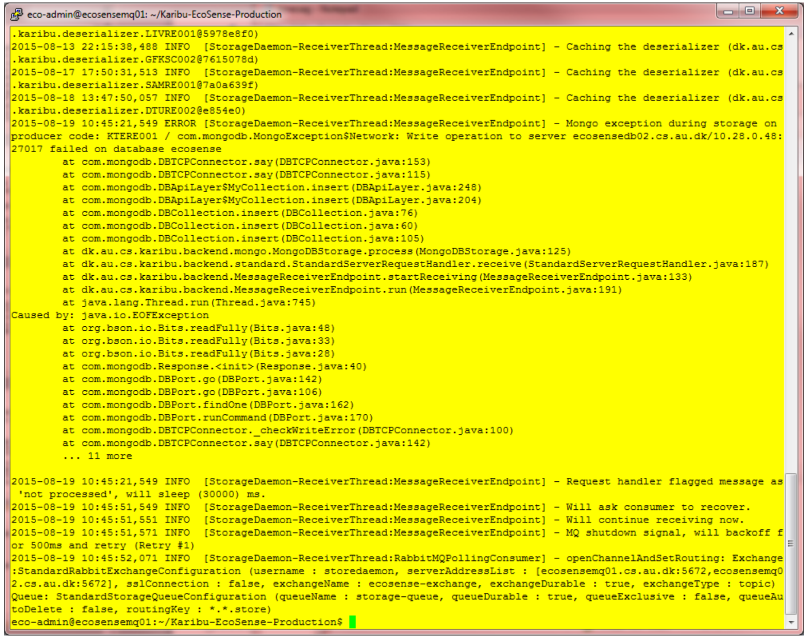
- *Steady state:* A system should be able to run indefinitely without intervention

- Otherwise, you get used to *fiddling*
  - *Which leads to what Nygard terms ”oh-no-second”*
    - The split second when you realize you have hit the wrong key, shut down the wrong server, deleted the wrong DB table…
    - My personal fear was ”db.GFKRE003.drop()”
      - 6TB of data from 4 years of data collection

AARHUS UNIVERSITET

My own Fiddling trick

- A linux shell looks like … a linux shell
- Make staging machines and production machines look **different!**
  - **Production machine** always have awful color choices!
  - I do not mistake one for the other!
    - I hope…



```
.karibu.deserializer.LIVRE001@5978e8f0)
2015-08-13 22:15:38,488 INFO  [StorageDaemon-ReceiverThread:MessageReceiverEndpoint] - Caching the deserializer (dk.au.cs
.karibu.deserializer.GFKSC002@7615078d)
2015-08-17 17:50:31,513 INFO  [StorageDaemon-ReceiverThread:MessageReceiverEndpoint] - Caching the deserializer (dk.au.cs
.karibu.deserializer.SAMRE001@7a0a639f)
2015-08-18 13:47:50,057 INFO  [StorageDaemon-ReceiverThread:MessageReceiverEndpoint] - Caching the deserializer (dk.au.cs
.karibu.deserializer.DTURE002@e854e0)
2015-08-19 10:45:21,549 ERROR [StorageDaemon-ReceiverThread:MessageReceiverEndpoint] - Mongo exception during storage on
producer code: KTERE001 / com.mongodb.MongoException$Network: Write operation to server ecosensedb02.cs.au.dk/10.28.0.48:
27017 failed on database ecosense
        at com.mongodb.DBTCPConnector.say(DBTCPConnector.java:153)
        at com.mongodb.DBTCPConnector.say(DBTCPConnector.java:115)
        at com.mongodb.DBApiLayer$MyCollection.insert(DBApiLayer.java:248)
        at com.mongodb.DBApiLayer$MyCollection.insert(DBApiLayer.java:204)
        at com.mongodb.DBCollection.insert(DBCollection.java:76)
        at com.mongodb.DBCollection.insert(DBCollection.java:60)
        at com.mongodb.DBCollection.insert(DBCollection.java:105)
        at dk.au.cs.karibu.backend.mongo.MongoDBStorage.process(MongoDBStorage.java:125)
        at dk.au.cs.karibu.backend.standard.StandardServerRequestHandler.receive(StandardServerRequestHandler.java:187)
        at dk.au.cs.karibu.backend.MessageReceiverEndpoint.startReceiving(MessageReceiverEndpoint.java:133)
        at dk.au.cs.karibu.backend.MessageReceiverEndpoint.run(MessageReceiverEndpoint.java:191)
        at java.lang.Thread.run(Thread.java:745)
Caused by: java.io.EOFException
        at org.bson.io.Bits.readFully(Bits.java:48)
        at org.bson.io.Bits.readFully(Bits.java:33)
        at org.bson.io.Bits.readFully(Bits.java:28)
        at com.mongodb.Response.<init>(Response.java:40)
        at com.mongodb.DBPort.go(DBPort.java:142)
        at com.mongodb.DBPort.go(DBPort.java:106)
        at com.mongodb.DBPort.findOne(DBPort.java:162)
        at com.mongodb.DBPort.runCommand(DBPort.java:170)
        at com.mongodb.DBTCPConnector._checkWriteError(DBTCPConnector.java:100)
        at com.mongodb.DBTCPConnector.say(DBTCPConnector.java:142)
        ... 11 more

2015-08-19 10:45:21,549 INFO  [StorageDaemon-ReceiverThread:MessageReceiverEndpoint] - Request handler flagged message as
 'not processed', will sleep (30000) ms.
2015-08-19 10:45:51,549 INFO  [StorageDaemon-ReceiverThread:MessageReceiverEndpoint] - Will ask consumer to recover.
2015-08-19 10:45:51,551 INFO  [StorageDaemon-ReceiverThread:MessageReceiverEndpoint] - Will continue receiving now.
2015-08-19 10:45:51,571 INFO  [StorageDaemon-ReceiverThread:MessageReceiverEndpoint] - MQ shutdown signal, will backoff f
or 500ms and retry (Retry #1)
2015-08-19 10:45:52,071 INFO  [StorageDaemon-ReceiverThread:RabbitMQPollingConsumer] - openChannelAndSetRouting: Exchange
:StandardRabbitExchangeConfiguration (username : storedaemon, serverAddressList : [ecosensemq01.cs.au.dk:5672,ecosensemq0
2.cs.au.dk:5672], sslConnection : false, exchangeName : ecosense-exchange, exchangeDurable : true, exchangeType : topic)
Queue: StandardStorageQueueConfiguration (queueName : storage-queue, queueDurable : true, queueExclusive : false, queueAu
toDelete : false, routingKey : *.*.store)
eco-admin@ecosensemq01:~/Karibu-EcoSense-Production$ 
```

Steady State

- Data purging
  - Remove old data from the DB
    - Can be pretty tricky in RDB
      - Referential integrity, orphaned rows, …
    - And perhaps even more so in NoSQL
  - Log files
    - If not purged, you run out of disk space
      - Java.io.IOException!
      - Dump a stacktrace in the face of the user ☺
    - Review the Log4J RollingFileAppender for non Docker use
    - For Docker, you write to StdOut which is stored ☹
      - Rewire to a logging system like ELK or Humio instead
        - » Which then run out of disk space ☺

Fail Fast

- *Fail fast:* Check resource availability at the start of a transaction, and fail immediately in case any is not available.

- If not, you waste CPU and human time doing stuff that will eventually have to be redone or undone

- Cook’s: *mise en place*
  - Find all ingredients before starting
    - Or the fish will burn while to try to find the chili paste…

Fail Fast

- Part of fail fast is also to validate human entered values as best possible before proceeding
  - Typically values entered in web form or similar
    - Avoid connecting the DB and do a query only to find that one of the query parameters were null…

- Fail Fast is a way to combat *slow response*
- Example:
  - CPF system will fail immediately if a key is not found in file.

Let it Crash

- *Let it Crash:* Create system-level stability, by sacrificing service-level(*) stability. The cleanest state your program has, is right after startup.
  - Nygard writes ‘component-level’, but guess that is synonym with our ‘services’.

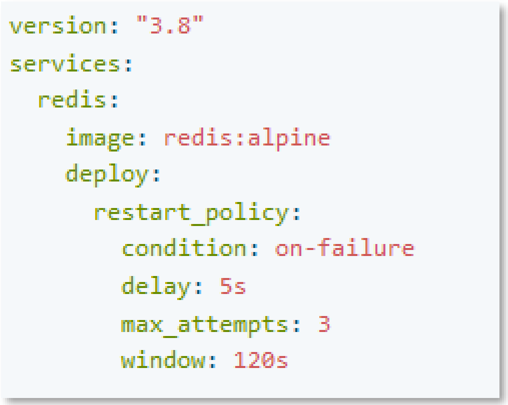
Inspired by the Erlang programming language

- Require
  - Limited granularity: isolate the crash to the service
    - Avoid cascading failures
  - Fast replacement: Only ‘let it crash’ if restarts are quick
  - Supervision: Don’t do local restarts, monitor on higher level
  - Reintegration: Consider how ‘back to work’ is orchestrated

Let it Crash

- We will return to HEALTHCHECK in Dockerfiles and compose-files.

- Allows
  - Granularity is the container
  - Restart by the swarm (restart time is within the ~seconds)
  - Simple supervision and reintegration is built-in in the swarm’s stack
    - Restart-policy



```
version: "3.8"
services:
  redis:
    image: redis:alpine
    deploy:
      restart_policy:
        condition: on-failure
        delay: 5s
        max_attempts: 3
        window: 120s
```

Handshaking

- *Handshaking:* Allows a client to assess whether a server has the capacity to answer a request, essentially providing the server the ability to tell the client to ”back off”
- Compare
  - Ping-echo and Heartbeat tactics
- Actually not implemented in most modern protocols, like HTTP, RMI, etc.
  - You will probably have to implement it yourself
  - Watch out for doubling the traffic: ‘can I do (1), then I do (2)’
    - Any HTTP call is expensive…

Test Harness

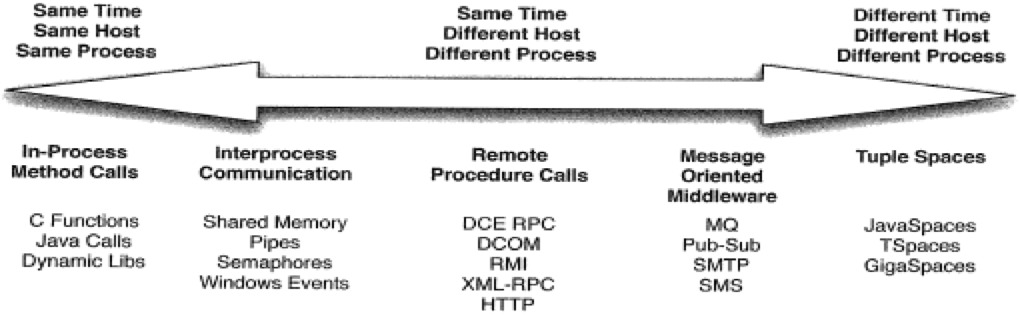
- *Test Harness:* A substitute for the remote end of an integration point that produce out-of-spec failures

  - Like
    - Refuse connection, accept and then die, packet loss
    - Slow responses, garbage responses, protocol errors
    - Send one byte every 30 seconds (DOS attack), send Megabyte long replies instead of Kilobyte long replies
    - Send HTML instead of XML, refuse all authentication credentials, …

Test Harness

- Da: Seletøj
- *Integrations test:* Testing that the parts are working together to identify integration failures
  - Obviously closely related to integration points and the large set of stability antipatterns
- Nygard: No – it is not strong enough
  - Integration tests test within-spec failure modes…
  - Operations excel in ***out-of-spec failure modes*** !!!
    - You do not get an error code from the remote server, it simply does not reply, or provide *slow response*

Decoupling Middleware



Same Time
Same Host
Same Process

Same Time
Different Host
Different Process

Different Time
Different Host
Different Process

In-Process
Method Calls

C Functions
Java Calls
Dynamic Libs

Interprocess
Communication

Shared Memory
Pipes
Semaphores
Windows Events

Remote
Procedure Calls

DCE RPC
DCOM
RMI
XML-RPC
HTTP

Message
Oriented
Middleware

MQ
Pub-Sub
SMTP
SMS

Tuple Spaces

JavaSpaces
TSpaces
GigaSpaces

Figure 5.4: Coupling Spectrum of Middleware

- Middleware decisions effect the implementation cost of systems significantly
  - Learn many architectural styles to ensure you pick the right one

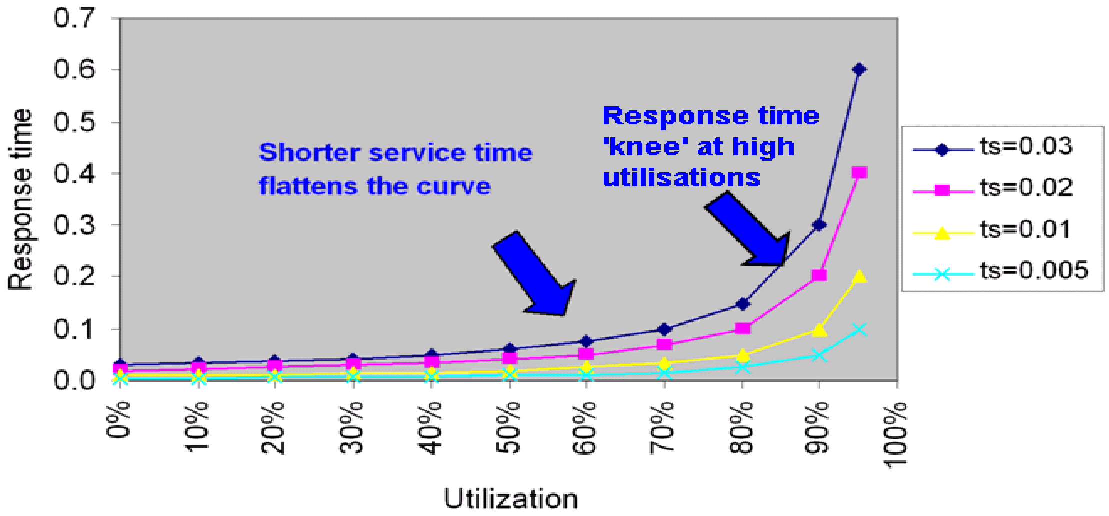
Shed Load

- *The world can always create more load than you can handle…*
  - There is no difference between ‘really, really slow’ and ‘down’.
- *Shed load:* Refuse new requests, if load gets too high.
  - Similar to ‘fail fast’ but you do not fail on service-level but on request-level
    - Not ‘time out exception’ but ‘503 service unavailable’
  - Guard calls to shed-loading services with circuit breakers or timeouts or …
  - Meaning of ‘Too high’?
    - Monitor own SLA to determine the tripping point

Shed = skille sig af med

Shed Load

- Back in the Queue Theory stuff…
  - Systems with ‘randomly timed requests’ will follow this distribution as the workload increase – response times increase exponentially ☹
  - Shed load when you hit the upper parts of the ‘knee’



Create Back Pressure

- *Every performance problem starts with a queue backing up somewhere…*
  - Little's law: L = lambda x R
    - L = number of requests in queue (Think Føtex kasse queue)
    - R = response time of request (Think 'time until I leave')
      - R = W + S, wait time + service time (in line + getting served)
    - lambda = arrival rate (arrival per sec, of custom.)
  - So
    - lambda is constant (influx of requests from the world)
    - If response time gets longer… (S large (dankortterminal i stykker)')
    - … then more requests in queue
    - … and then queue eats all memory => crash

Create Back Pressure

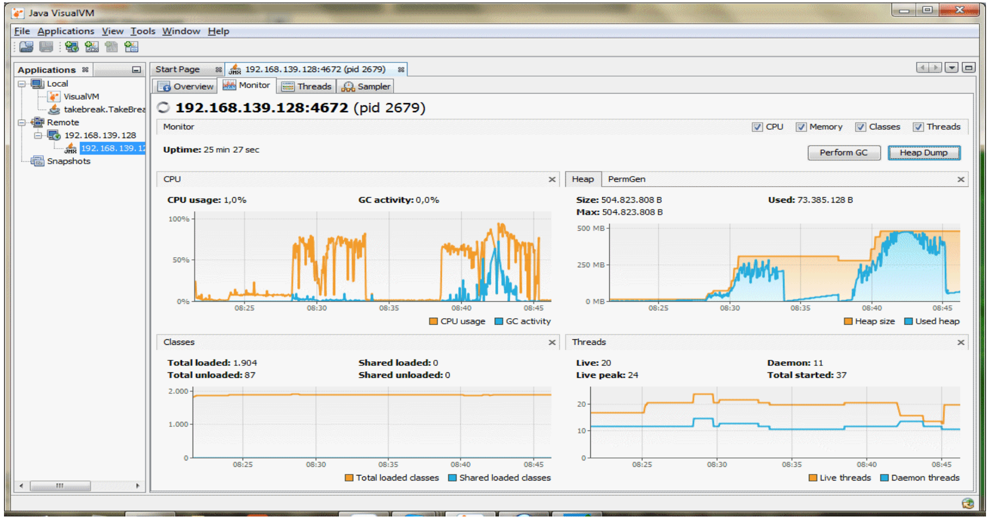
- So
  - … we do not want *unbounded queues!*
- Bounded queue, what to do with 'out of space'?
  - Pretend to accept new item, but actually drop it
  - Accept new item, drop something else
  - Refuse the item (= shed load)
  - Block request (producer) until there is room in queue
- Dropping options
  - In many real-time systems, only latest reading is interesting
    - Aircraft flight control – who gives a damn about that angle the rudder was in 30 seconds ago – more interesting what is *now*

Create Back Pressure

- Block producer option
  - Introduces ‘flow control’, applying ‘back pressure’ upstream
    - (probably) propagates all the way back to the client, who will be throttled down until queue releases…
- *Create Back Pressure:* Use finite queues and block producers if queue overflows, to slow down instead of crashing [Own definition]
  - Alternative to ‘shed load’
  - May lead to ‘blocked threads’ obviously
    - Are we crashed or just extremely slow?
  - Only use ‘within system boundaries’
    - Use ‘shed load’ across system boundaries instead, like open internet

Ex

- My Lyon airport load test
  - MongoDB was slow, so node memory exhausted



192.168.139.128:4672 (pid 2679)

Uptime: 25 min 27 sec

CPU usage: 1,0% **GC activity:** 0,0%

Size: 504.823.808 B **Used:** 73.385.128 B
Max: 504.823.808 B

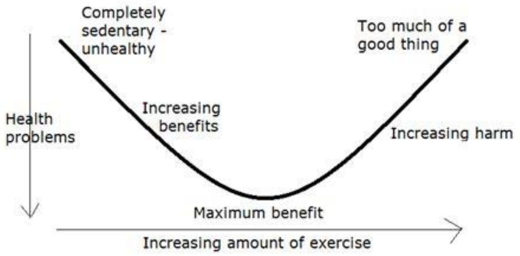
Total loaded: 1.904 **Shared loaded:** 0
Total unloaded: 87 **Shared unloaded:** 0

Live: 20 **Daemon:** 11
Live peak: 24 **Total started:** 37

- Solution
  - I set the ‘prefetch’ to 1!
    - Now Java connector does not fetch next until MongoDB has stored it (acknowledge message)
    - Thus, the RabbitMQ itself acts as Queue, rather than the consuming node’s heap acting as queue!
    - Thus *applying back pressure upstream until I hit a service that can actually handle the pressure…*

Governor

- Steam engines: They can run so fast, they will break!
  - Engineering device: Put a speed limit on: the ‘governor’
- *Govenor:* Stateful and time-aware control plane logic that prevents a system from exceeding its safe limits. Actions within safe limit should be fast, outside increasing resistance must be applied
  - *Shutting down instances is unsafe, deleting data is unsafe …*
  - Antidote for the ‘force multiplier’ antipattern…
  - Safe limits is kind of ‘U-shaped curve’
    - Slow actions down if moving outside the buttom of the U curve…



Bounded Result Sets

- Nygard does not mention a pattern to combat *unbounded result sets*
  - Should be reasonably obvious…

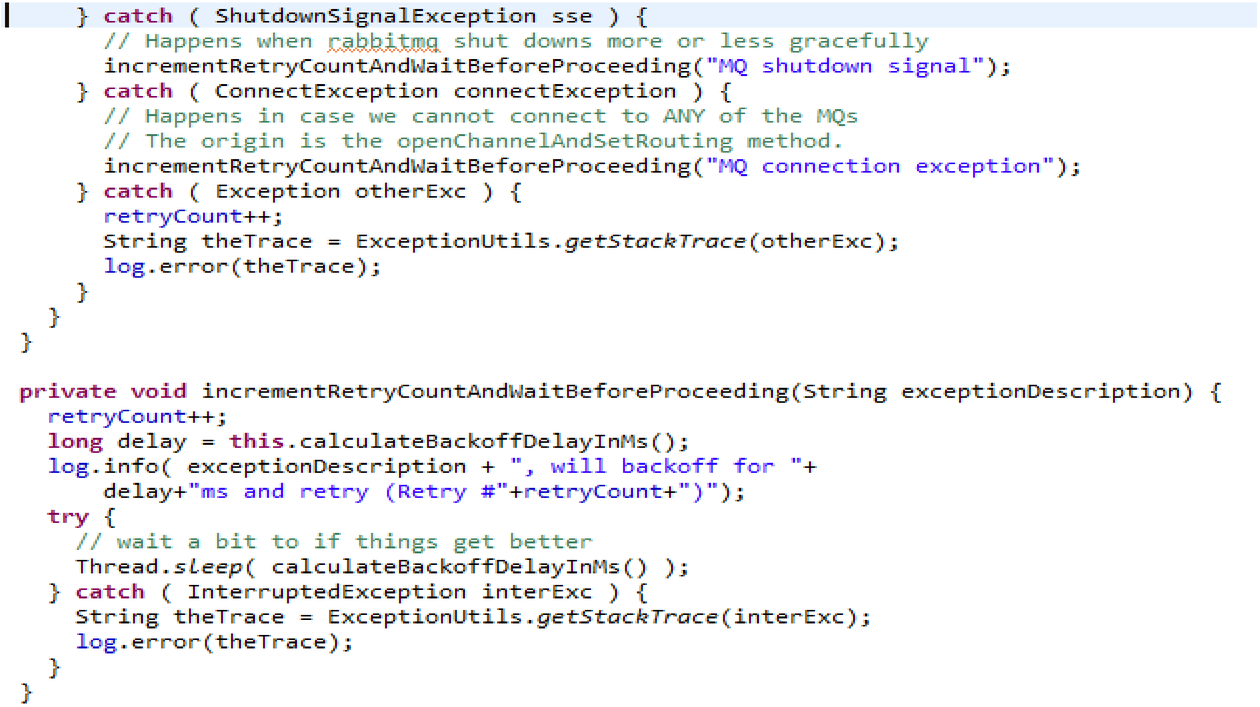
- *Bounded Result Sets:* Return large results sets in *chunks* that can be *iterated.* (Pagination)
  - *Not ”get bible”, but ”get bible, page 7564, page 7565, …”*
  - *Mongo: use find().skip(n*page).limit(n)*

Retry

- Perhaps too obvious, but…
- Why does Nygard not mention ‘Retry’ as a pattern?
  - Only indirectly in ‘time outs’ pattern
- *Retry:* If a request fail/time out, then retry the request some time later a number of times before giving up
  - Only do it, if it makes sense! And who does the retry?
  - Fixed retry intervals
    - Beware: May lead to dogpiles! We herd the calls together in lumps
  - Gaussian retry intervals (?)
  - ‘Exponential backoff’
    - Retry after 1s, 2s, 4s, 8s, 16s, and then give up.

Use Timeouts

- Example
  - Exponential Back-off
    - Wait 1 second, then 2, then 4, then 8, then 16…



```
    } catch ( ShutdownSignalException sse ) {
      // Happens when rabbitmq shut downs more or less gracefully
      incrementRetryCountAndWaitBeforeProceeding("MQ shutdown signal");
    } catch ( ConnectException connectException ) {
      // Happens in case we cannot connect to ANY of the MQs
      // The origin is the openChannelAndSetRouting method.
      incrementRetryCountAndWaitBeforeProceeding("MQ connection exception");
    } catch ( Exception otherExc ) {
      retryCount++;
      String theTrace = ExceptionUtils.getStackTrace(otherExc);
      log.error(theTrace);
    }
  }
}

private void incrementRetryCountAndWaitBeforeProceeding(String exceptionDescription) {
  retryCount++;
  long delay = this.calculateBackoffDelayInMs();
  log.info( exceptionDescription + ", will backoff for "+
      delay+"ms and retry (Retry #"+retryCount+")");
  try {
    // wait a bit to if things get better
    Thread.sleep( calculateBackoffDelayInMs() );
  } catch ( InterruptedException interExc ) {
    String theTrace = ExceptionUtils.getStackTrace(interExc);
    log.error(theTrace);
  }
}
```

Who does Retry?

- Exercise:
  - PlayerServant delegate to CaveStorage delegate to MongoDB
  - MongoDB’s primary fails and throws exception…
    - We will talk about passive replication technique shortly
  - Exception caught in CaveStorage but
    - Handle it locally in CaveStorage and do a retry?
    - Rethrow as ‘ElectionException’ and handle in PlayerServant?
  - Pro and Con of each solution?

Experience F2020

- In ‘design for failure’ in F2020 I herded students into trying to find graceful degradations
  - Like retries
- General impression
  - Seemed that the code became cumbersome and the results for the users not quite intuitive
    - “dig n My new room”
      - “Could not dig your room, will retry later” ???
- So this year
  - *Fail fast and report it*
    - *“Could not dig your room, you have to try again later”*

Summary

- Phew…

- Plus
  - Shed load
  - Back pressure
  - Govenor

  - Bounded Result sets
  - Retry

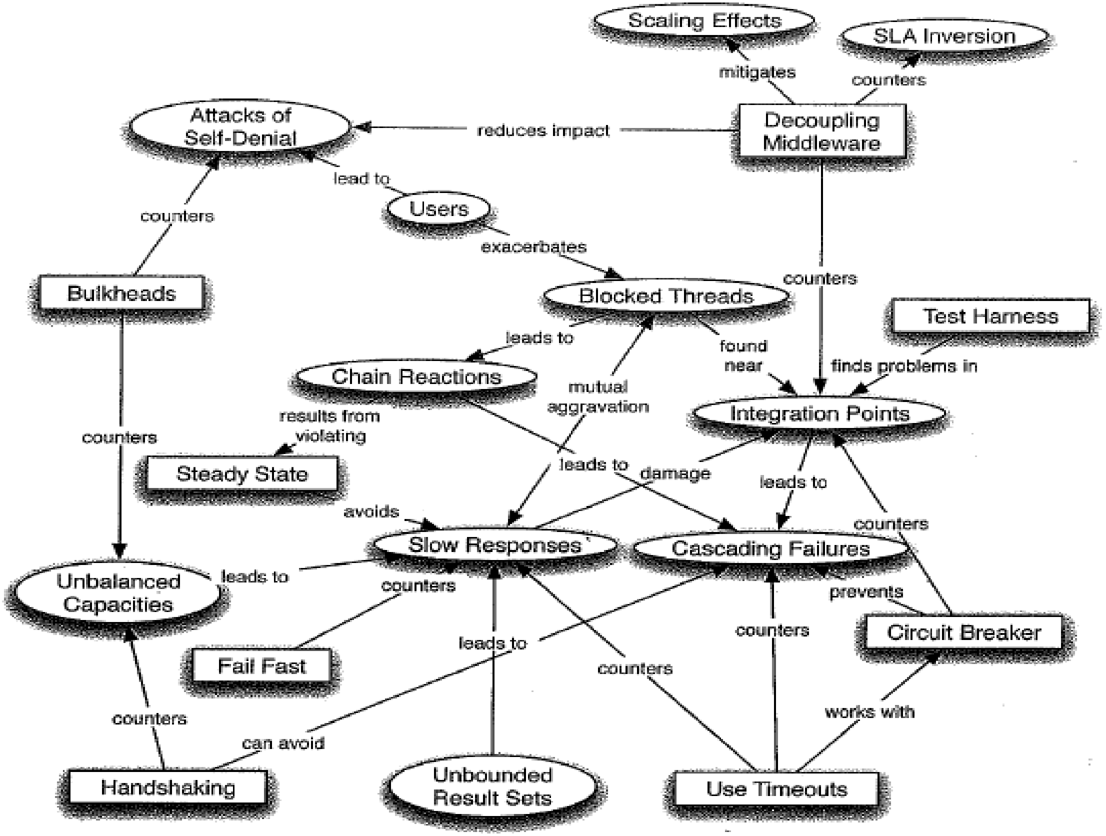


Figure 3.1: Interaction of Patterns and Antipatterns