Erlang 24/7
Chalmers 2014-02-26

Cons T Åhs
Keeper of The Code
cons@klarna.com
@lisztspace

Cons T Åhs

- Keeper of The Code at klarna
  - Architecture - The Big Picture
  - Development - getting ideas to work
  - Code Quality - care about the details
  - Increase competence of developers - better developers are more productive and delivers better solutions

Klarna

Cons T Åhs

- Student in Uppsala 1981 - 85 (writing code for five years, Lisp before starting in Uppsala)
- Uppsala University (1985-2000, 2002, 2009, 2012), research & teaching; foundations, algorithms, functions, relations, objects, compilers, pragmatics, theory, theorem proving, formal program correctness..
- Consultant (1991-); online poker, low level networking, medical imaging, graphics, finance, musical notation (Lisp), speech synthesis (Lisp, Prolog), compilers (Lisp, Prolog), real time video decoding, teaching..
- Klarna from Feb. 28, 2011

Klarna

Klarna - The Business

- Make shopping on the net simpler, safer, more fun.
- Pay by invoice after the goods are delivered
- Customer checks out at estore
  - Klarna identifies customer and investigates credit
  - estore sends goods and invoice
  - Klarna pays estore (Klarna takes the risk)
  - customer pays Klarna
- Identification and risk determination needs to be done fast (a few seconds)

Klarna - The Business

- Klarna makes money by taking a calculated risk
  - We're buying and selling invoices
- What do we need to be good at?
  - Identifying our customers - do you exist?
    - Interesting algorithms, lots of data
  - Evaluate risk - are you going to pay?
    - Interesting algorithms, lots of data
  - Bookkeeping - we're essentially a bank.
    - Only lots of data, no algorithms

Klarna

Klarna - The Facts

- Founded in 2005
- Revenue doubled every year from start - we’re growing exponentially
- Sweden, Norway, Denmark, Finland, Germany, Netherlands, Austria - more to come..
- Over 800 employees
- Over 15K estores and growing

Very nice for the shareholders..

Klarna - The Facts

- Currently over 2.5M transactions/month
  - average ≈ 1/s, but there are peaks both over the day and the year
  - you can't build for the average
- People shop all the time
  - Available 24/7 - no downtime
  - Software upgrades with no downtime
  - Hardware upgrades and relocation with no downtime

.. we need to build and maintain a robust system

Klarna

Klarna - The technology

- Use technology from a domain with similar needs
  - large amount of messages/transactions
  - high demands on availability, scalability, robustness
  - low tolerance for downtime for software or hardware upgrades
- We're using Erlang/otp
  - "functional," state is handled in processes
  - easy to distribute and communicate
  - can handle large amounts of processes
  - robust
  - soft real time

Klarna

Erlang
A functional language

- Dynamically typed functional language
- No side effects; variables are bound once and the value can not be changed
- Every expression computes a value
- Pattern matching provides parallel binding and compact programs (mixed blessing - beware!)
- Looks very much like Prolog
- The power of higher order functions and closures
- It is easier to understand, reason about, compile and debug a functional program.

Klarna

24/7 implies interesting challenges

- 24/7 means having a goal of 100% availability
- 99.99% availability translates to about 4 minutes of allowed downtime/month
- There is not too much you can do in 4 minutes
- No planned shutdown, even for upgrades
- Code and data format change without downtime
- Subsystem switching without downtime
- Hardware switching without downtime
- etc..
- There is no notion of stopping and starting the system - it just runs..

Klarna

Joe Armstrong says:

Each year your sequential programs will go slower.

Each year your concurrent programs will go faster.

Concurrent and distributed programming

- With concurrent programming troubles form when you have a shared and mutable state.
- Problem typically solved by using synchronisation with locks
  - Complicated - you have to know when to lock
  - Can lead to more problems - performance degradation
  - Cooperative model - all parts of the program must agree
- Take away one and your on safe ground.
- Erlang takes away both!

Klarna

No shared state, no mutable state

- Each process has a state of its own, or rather a sequence of states; possibly a new state after receiving a message
- Each process has a private heap
- Each process has a message queue (the implementation handles these)
- Processes can not share state, even when they live in the same VM.
  - All communication must be done with messages.
  - Asynchronous message passing - messages are <u>copied</u> between processes

Klarna

No shared state

- Why?
  - Background (telecom switches) with a large number of small and short lived processes
  - When a process dies there is no risk reclaiming the whole process
  - No other process can access the memory it used
  - Nothing happens if you send a message to a dead pid
  - The dead process can not reference the memory of another process
  - Leads to robustness

Klarna

Erlang
built for fault tolerance

- No shared state means that a crashing process will not take another process down.
- A crashing process can notify another process, which knows how to restart.
  - Processes can be linked with each other, thereby creating process and supervisor structures
- Ease of distribution and horisontal scalability also makes it easy to build redundant systems - we have immediate failover if a server dies.
  - Not trivial, but support for it exists in the language

Build it robust

- Accidents happen
  - “this can never happen”
  - unexpected input
  - missing case etc
- Be prepared!
  - Don’t assume your program will never crash
  - Limit effects of a crash - Erlang does this for you
- Note:
  - Exceptions are for exceptional cases

Klarna

Processes everywhere

- Processes in Erlang are cheap and flexible
  - creation and destruction is fast and easy
  - initial size is small, typically just hundreds of bytes
  - they can grow surprisingly large..
- A typical system will consist of a (large) number of communicating processes
- At any point, one of them can have a mishap and die
- A dying process screams out in agony
- Catch the death and act accordingly
  - Restart a process that dies
  - Restart other parts of the system

Klarna

Linking Processes

- Simple creation of processes is done using `spawn/1`
- There is also `spawn_link/1`
  - works like `spawn/1`
  - creates a link between the calling process (`self()`) and the newly created process
  - We're saying that these processes are important to each other.
  - If one dies the other dies as well.
  - Links are symmetrical
  - A process can be linked to several other processes, thus building process hierarchies

Klarna

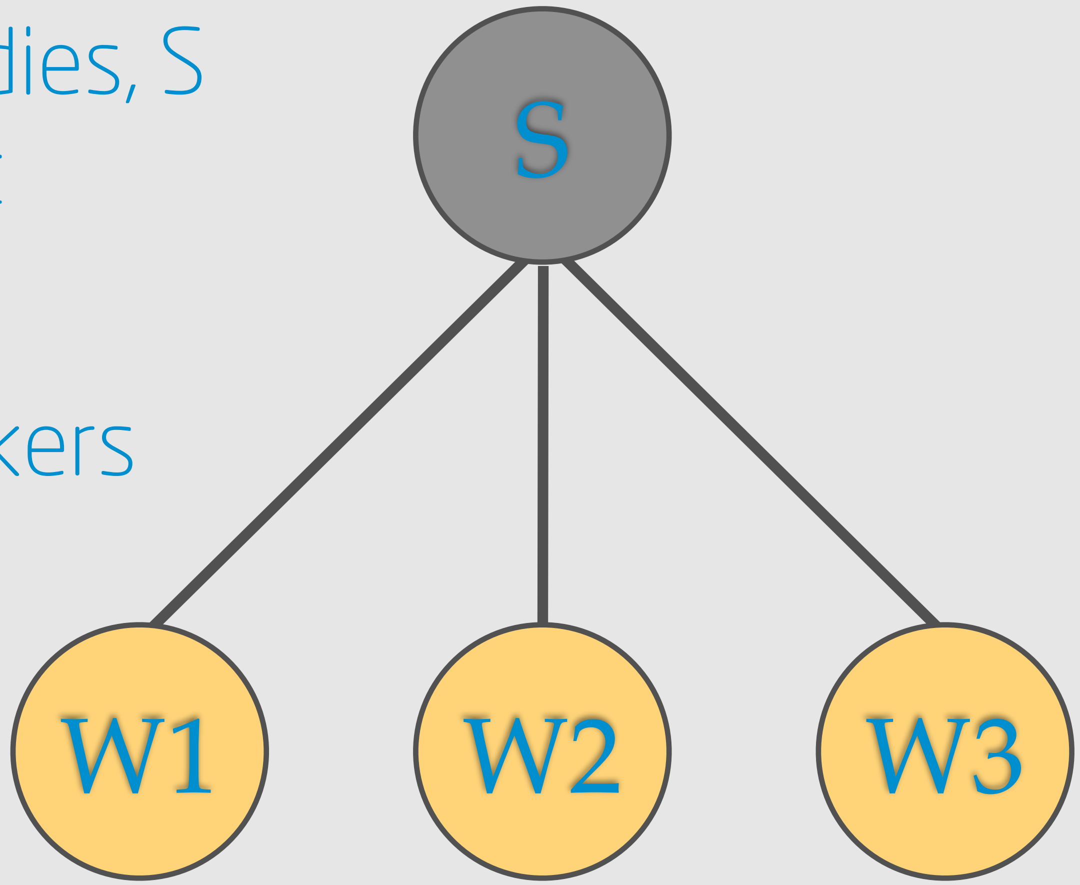
Catching the death of a process

- Instead of having process groups dying together a process can catch the death of other processes it is linked to
- Call `process_flag(trap_exit, true)`
  - If a linked process dies, instead of getting an exit signal that would kill the process, an ordinary message of the form `{‘EXIT’, From, Reason}` is received
  - The message can be processed by an ordinary `receive ... end` in a suitable manner

Klarna

Workers vs Supervisors

- Trapping exit signals is asymmetric
  - S is linked to W1, W2, W3
  - S traps exits
  - If S dies, W1, W2 and W3 will die
  - If either of W1, W2, W3 dies, S will get to know about it
  - S is a supervisor
  - W1, W2 and W3 are workers



!DIY

- The low level mechanisms are few and simple
- Putting them together and getting it Right (tm) is tricky (and distracts you from your core task)
- Use the supervisor behaviour (very similar to an interface in Java)
- The supervisor behaviour states that you implement one function `init/1` which should return a term

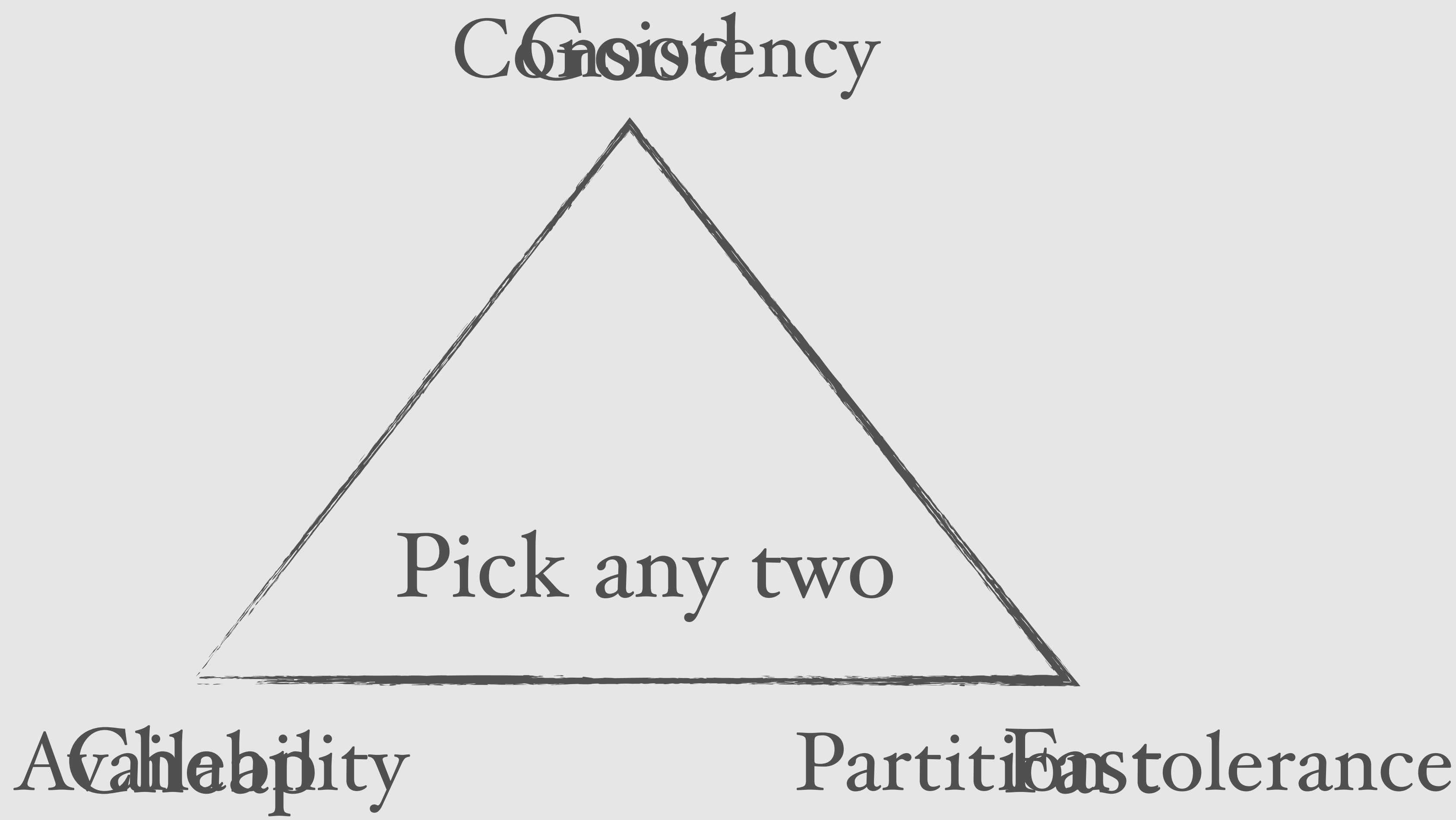
```
{ok,
 {RestartStrategy, MaxRetries, Time},
 [ChildSpec]}
```

Restart Strategies

- Allow a maximum of `MaxRetries` during `Time` seconds. If more are needed, the supervisor is terminated (and possibly handled by another supervisor)
- `one_for_one` - restart children independently of each other
- `one_for_all` - if one dies, restart all
- `rest_for_one` - if one dies, restart all “after” that one
- `simple_one_for_one` - like one_for_one, but all workers run the same code and can be added dynamically

Klarna

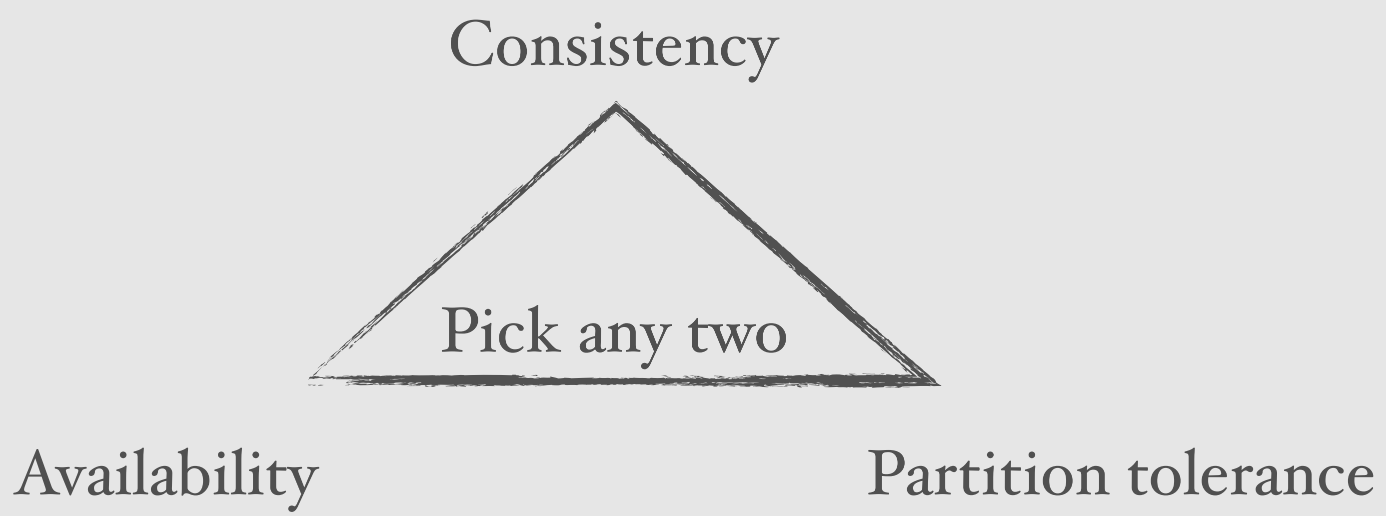
Brewer's CAP Theorem



Consistency - all nodes see the same data at the "same time"
Availability - clients receive answers "immediately"
Partition tolerance - service operates despite message loss between nodes

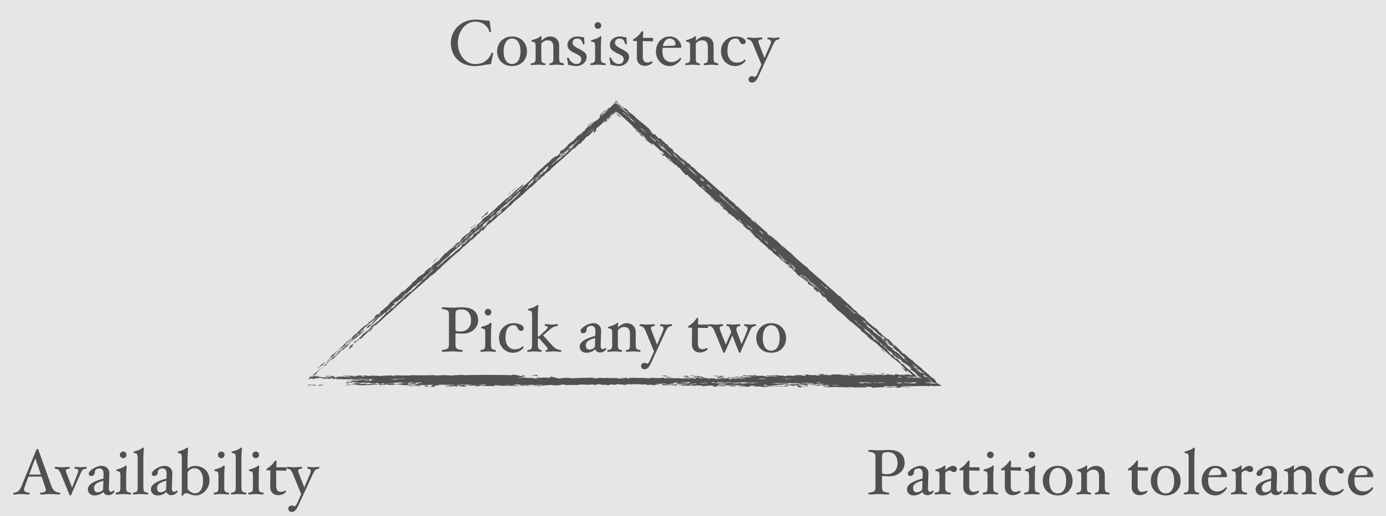
When your network partitions, your distributed system will be either consistent or available. At best.

Brewer's CAP Theorem



- Let the business decide which one to sacrifice
- Which two traits are crucial for making money?
- Three types of systems - examples?
  - Consistent and available
  - Available and partition tolerant
  - Consistent and partition tolerant

Brewer's CAP Theorem



- We handle money and risk
  - Consistency is definitely important!
- We want customers to spend money
  - We have to be available!
- Good bye, partition tolerance..?

Klarna

Theoretical Dilemma Meets Reality

- The problem:
  - Construct a system that is consistent, highly available and can scale when we grow exponentially
  - Scalability is often solved by growing horisontally, i.e., by adding more nodes (thus exposing us to risk of partioning).
- The reality [time to market is important]:
  - We don’t know we’re going to grow exponentially - focus on current problems; ignore the future.
  - This is reality - solve problems when they come along with simple real world solutions. Theory..?

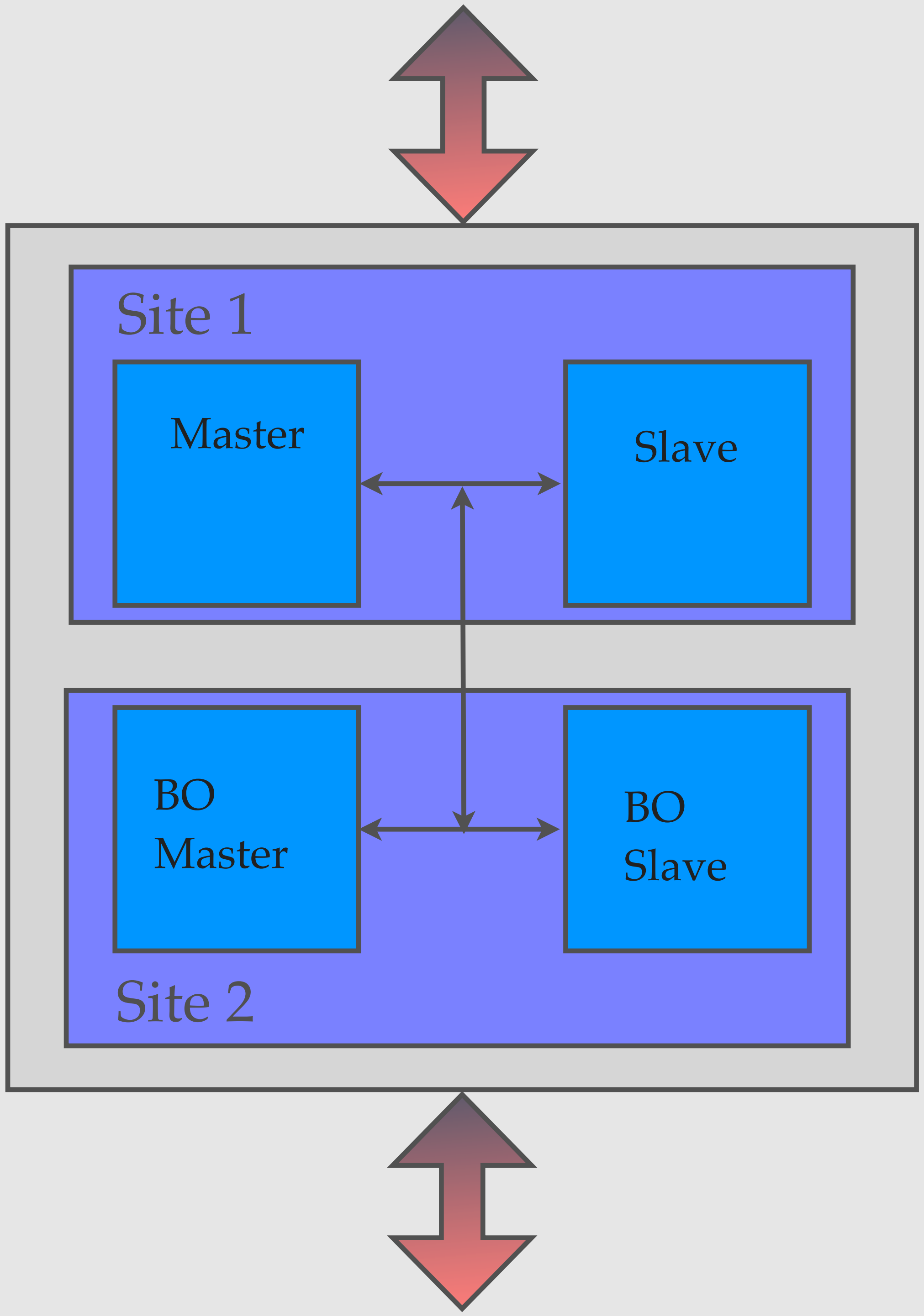
Klarna

The First System

- Focus on consistency and availability
  - Use several nodes for fail over
    - When, not if, a node dies another one can take over
  - Replicate data between nodes
    - Data is consistent between nodes and safe
  - Know what you’ve done
    - Logs can be used to rebuild state after failures

Klarna

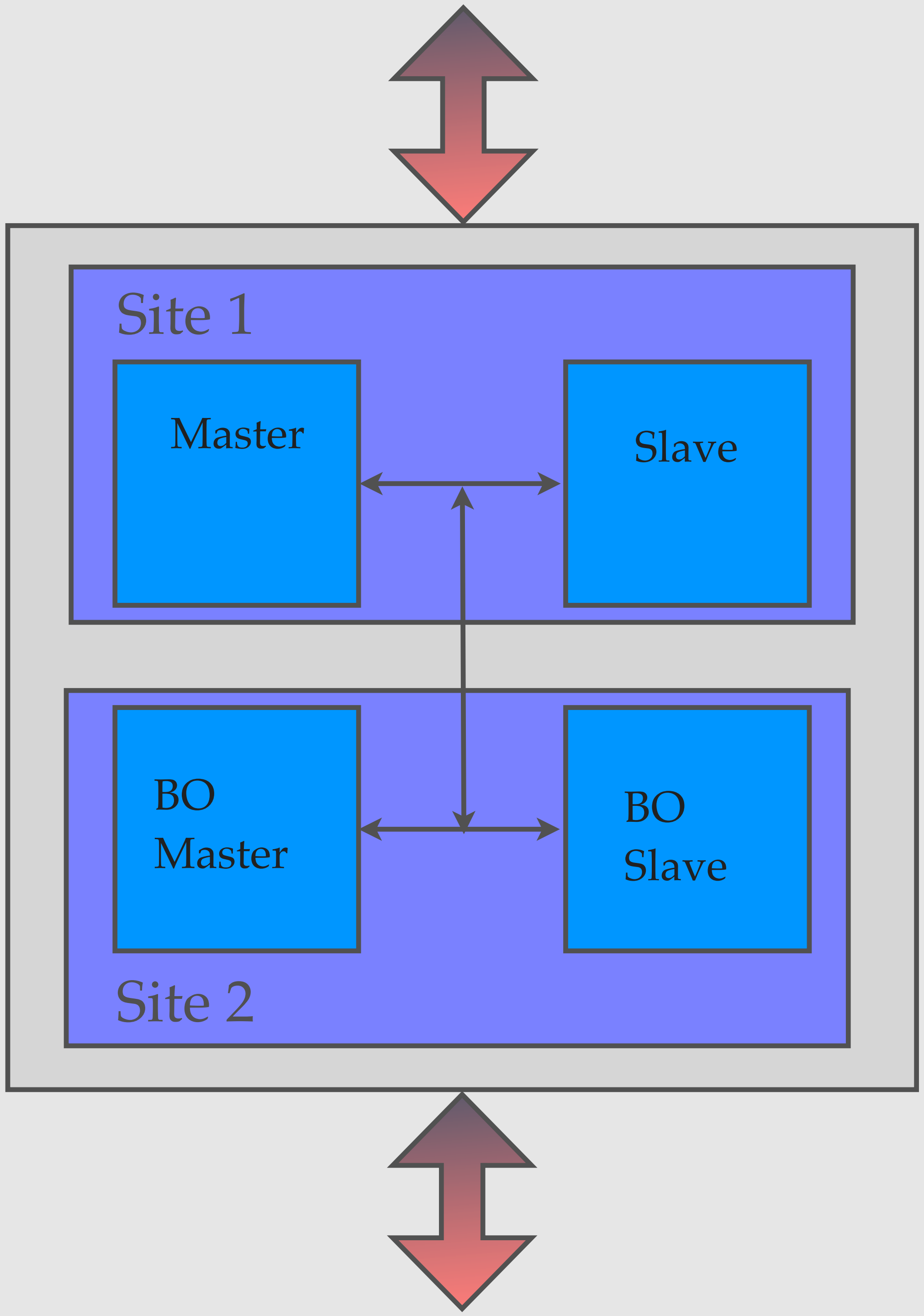
The First System



- Each node runs erlang
- Mnesia is used for persistence
- Master and slave have different main tasks
  - master handles incoming purchases
  - slave handles web traffic
- Back office (BO) handles customer service
- Data is replicated among the nodes, so all nodes have the same data
- If one node dies, the other takes over all responsibilities until the dead one comes up again (master and slave might then switch)
- Transaction logs are kept to rebuild state if needed
- Regular backups are kept

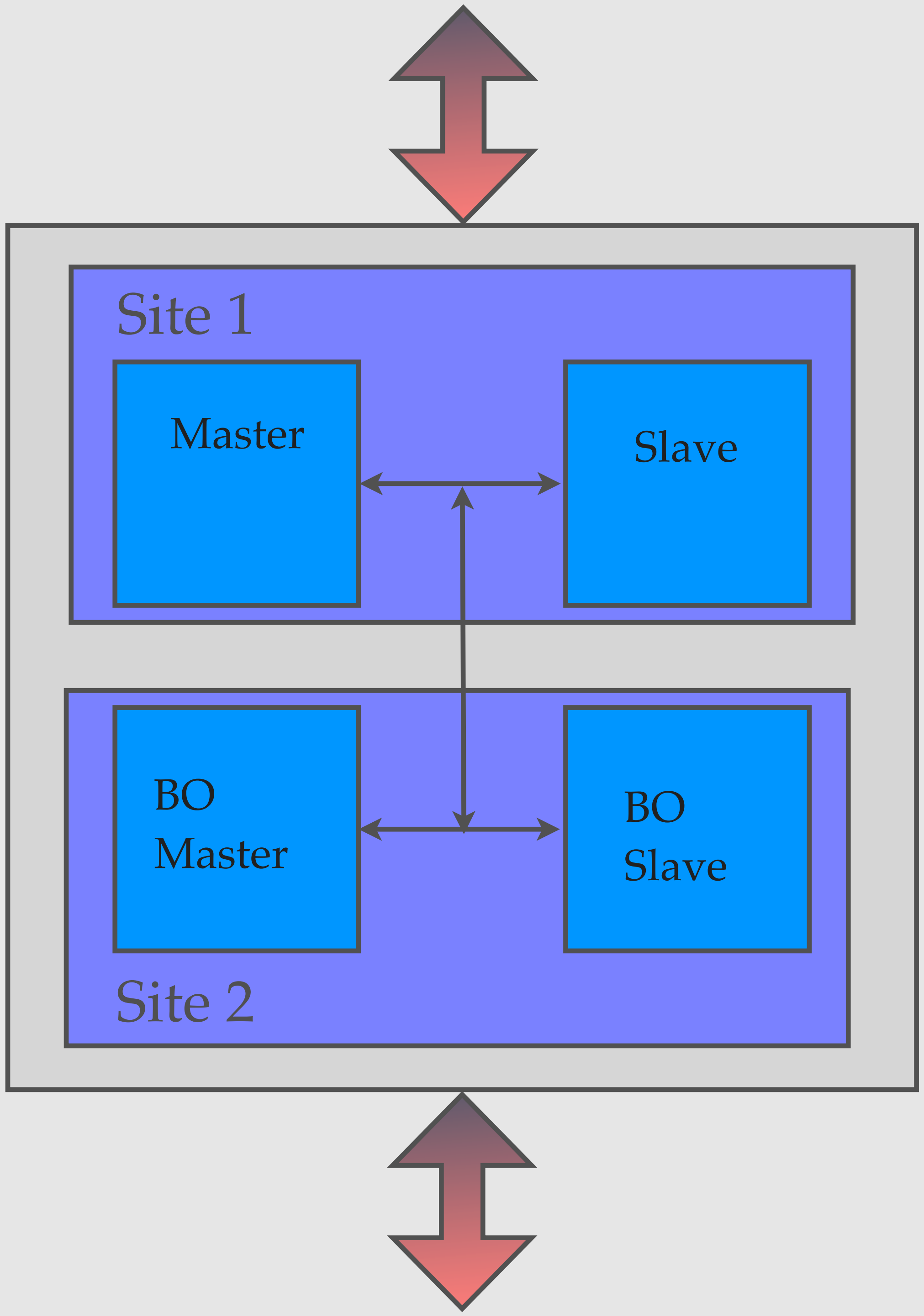
Klarna

The First System



- Availability is covered by multiple nodes being able to fail over fast; this is immediate.
- Consistency is covered by letting the nodes/sites have different responsibilities and replicating data.
- Partition tolerance is handled in the same way as a dead node, i.e., a node will take over all responsibilities in a site. Data is synced upon contact again.
- We can keep availability even when external services don't answer.
- When one node has died, we're vulnerable during the time the other takes to restart.
- If both nodes die on a site, we have a serious problem.

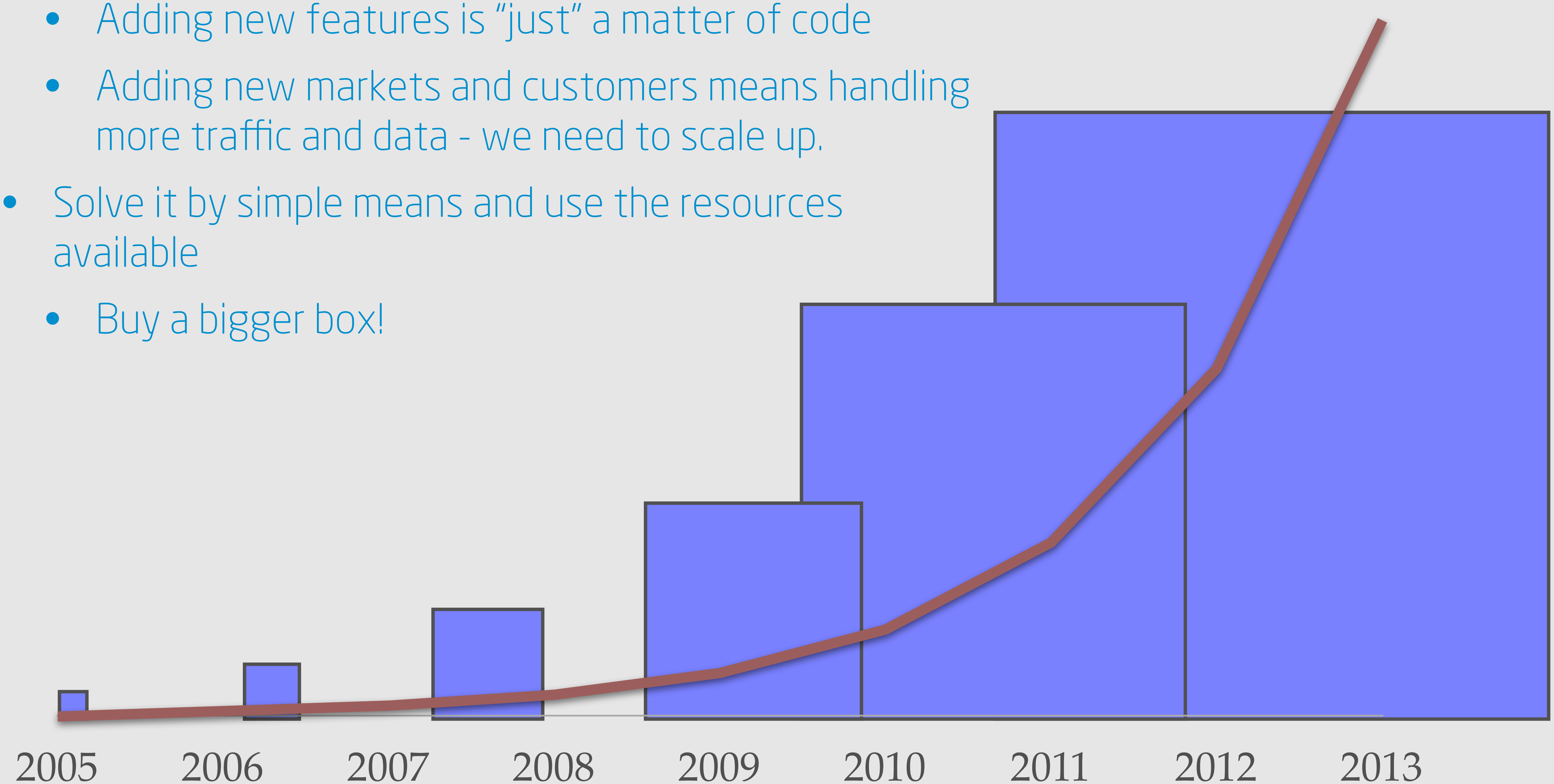
The First System



- Software upgrades are done several times each week without downtime or stopping the system.
- Code is loaded while normal traffic is flowing.
- OS, erlang and hardware can be upgraded without downtime; stop one node, do maintenance and restart.
- Hardware can be moved without downtime; stop one node, move it and restart.
- This has served us well from the start and is still doing quite nicely.
- Growth has been exponential, but this has also been able to handle the needed scaling.
- How?

Solve Problems When They Arise

- Success means growth
  - Adding new features is "just" a matter of code
  - Adding new markets and customers means handling more traffic and data - we need to scale up.
- Solve it by simple means and use the resources available
  - Buy a bigger box!



Klarna

Theory is catching up

- We can't scale like that forever
  - Data sets will be extremely large
  - We're getting closer to the practical limits of certain aspects erlang/otp; it wasn't really designed for large scale systems
  - Traffic increase, both incoming and between nodes will expose bottlenecks
  - Moving into new markets might mean having nodes closer to the market, either for latency or regulatory reasons
- Are we going to build a second system?
  - No, the Second System Syndrome is Real.
  - Change imposes risk and we need to maintain availability
  - Remodel the existing system to be able to scale better by true horisontal scaling.
  - Rebuild and replace components incrementally

Klarna

New Directions

- Scale horisontally by having multiple front end nodes, handling purchases and external web traffic.
- Move towards SOA, with stateless services for identification, risk assessment, payment handling.
- The front end has high demands for availability and scalability; high SLAs.
- Scale vertically by separating a back end to a more traditional transaction oriented bank like system.
  - Less critical.
  - We can tolerate inconsistency between front and back end. Availability (FE) is more critical.

Klarna

New Directions

- Availability still high priority.
- Consistency might suffer when moving purchase nodes apart, but data will be consistent eventually.
- Fail over between purchase nodes, when a whole purchase node fails
- Partition tolerance needs to be handled within a purchase node between services.
- Messages to back end handled by a message queue and handled in order. Less critical.
- This is currently work in progress.

Klarna

Interested in Klarna?

- Starting points
  - engineerng.klarna.com
  - klarna.com/jobs
  - recruitment@klarna.com