Everything you wanted to know about architecting a system but were afraid to ask……

OSCON
Architecting for Reliability & Scalability
Austin 2016
HELLO!
Francesco Cesarini
francesco@erlang-solutions.com
www.erlang-solutions.com
@francescoC
Chapters 3 - 12

- Ch 3: Behaviours
- Ch 4: Generic Servers
- Ch 5: Controlling Behaviours
- Ch 6: Finite State Machines
- Ch 7: Event Handlers
- Ch 8: Supervisors
- Ch 9: Applications
- Ch 10: Special Processes
- Ch 11: Release Handling
- Ch 12: Software Upgrade
Chapter 13

Ch 13: Node Architecture
Chapter 13

- Ch 13: Distributed Architectures
- Ch 14: Systems That Never Stop
- Ch 15: Scaling Out
- Ch 16: Monitoring and Preemptive Support
- TUTORIAL OBJECTIVES -

Break down the task of architecting your system into 10 easy steps, understanding the tradeoffs you have to make between reliability, scalability and data consistency.

▸ Build systems with no single point of failure.
▸ Understand the limits of consistency and reliability when scaling your system.
▸ Understand the tradeoffs you need to make when designing your system.
- CONTENTS -

- The Past
- Distributed Architectures
- Systems That Never Stop
- Scaling Out
- Monitoring and Preemptive Support
The Past

As hardware, network and consensus protocols evolve, so do the distributed architectures built around it.
- AXD301 Switch 1996 -

- A Telephony-Class, scalable (10 – 160 GBps) ATM switch
- Designed from scratch in less than 3 years
- AXD 301 Success factors:
  
  Competent organization and people
  Efficient process
  Excellent technology (e.g. Erlang/OTP)
- AXD301 Switch 1996 -

Active

Stable-state replication

Standby

Device board

Control signalling

Data path

Control plane

User plane

Stable-state replication
Jabber Proxy 2000

- de-multiplexing
- state/error handling
- multiplexing
- sockets
- listeners
- simple 1-1
- users
- sockets
- supervisor
- Third Party SMS Gateway 2004 -

Diagram:

- Third Parties
  - HTTPS
  - HTTP

  Nortel iSD-SSL
  - HTTPS
  - HTTP

  Public node (yaws)

  Transaction node
  - HTTP
  - Logs db (1 TB)

  SMSC
  - MO-SMS
  - MT-SMS

  Prepay Db

  Reply node
  - HTTP
  - Real time cust db
get/put("artist", "REM", R/W=2)

{ok, Object}
Distributed Architectures

A node is the smallest executable standalone unit consisting of a running instance of the Erlang runtime system.
- **NODE TYPES** -

1. Split up your system's functionality into manageable, stand-alone nodes:

   - Identify Services
   - Isolate Failure
   - Memory vs CPU bound
   - Safe vs Unsafe Nodes
2. Decide what distributed architectural pattern you are going to use:

- Fully Meshed
- Dynamo Approach
- SD Erlang
- Service Oriented Architecture
- Peer to Peer
- Sharding
2. Decide what distributed architectural pattern you are going to use:

- Fully Meshed
- Dynamo Approach
- SD Erlang
- Service Oriented Architecture
- Peer to Peer
- Sharding
a ring with 32 partitions

2^{160}/2

2^{160}/4

0

hash(<<"artist">>,<<"REM">>)

node 0
node 1
node 2
node 3

Dynamo
Dynamo Style Scalability Reaching 100 Nodes:

- Consistent Hashing
- Virtual Nodes
- Gossip Protocol
- Hinted Handoff
- Sloppy Quorums
- Scalable Distributed Erlang -
3. Choose the network protocols your nodes, node families and clusters will use when communicating with each other:

- Distributed Erlang
- MPI, 0MQ, UDP, SSL
- Sockets or SSL
- REST, AMQP, SNMP, XMPP, MQTT
4. Define your node interfaces, state and data model:

- Use stories
- Validate your choice of Node functionality
- Reduce communication
- Reduce Data Redundancy
- Standardize APIs
STEPS EVOLVING AROUND DISTRIBUTION

1. Split up your system's functionality into manageable, stand-alone nodes.
2. Decide what distributed architectural pattern you are going to use.
3. Decide what network protocols your nodes, node families and clusters will use when communicating with each other.
4. Define your node interfaces, state and data model.
Systems That Never Stop

You need at least two computers to make a fault tolerant system.
Sharing Data

You have at least two computers to make a fault tolerant system, you need to share state and data.

Buy train set ➔ Load Balancer ➔ Buy ➔ Web ➔ Buy ➔ Logic ➔ Session1: book, train set

Remove ➔ Load Balancer ➔ Remove ➔ Web ➔ Remove ➔ Logic ➔ Session1: train set
Network Partitions
1 Request

Client

Web

Logic

Request

Reply

2 Request

Client

Web

Logic

Request

Reply

{duplicate, Reply}
5. For every interface function in your nodes, you need to pick a retry strategy:

- At most once
- At least once
- Exactly Once
Handling Errors
- DATA DUPLICATION STRATEGY -

6. For all your data and state, pick your sharing strategy across node families, clusters and types, taking into consideration the needs of your retry strategy:

- Share Nothing
- Share Something
- Share Everything
Trade-offs

- **Consistency**
  - exactly once
  - at least once
  - at most once

- **Availability**

- **Reliability**

- **Recovery Strategy**

- **Sharing Data**
  - share everything
  - share something
  - share nothing
STEPS EVOLVING AROUND AVAILABILITY, CONSISTENCY & RELIABILITY

5. For every interface function in your nodes, you need to pick a retry strategy.

6. For all your data and state, pick your sharing strategy across node families, clusters and types, taking into consideration the needs of your retry strategy.

Reiterate through steps 1, 2, 3, 4, 5 & 6 until you have the trade-offs which suit your specification.
Scaling Out

Distribute for scale and replicate for availability.
Scaling Vertically
Scaling Horizontally

web server  web server  web server  web server  web server  web server  web server  web server
Horizontal scalability is a tradeoff between consistency and availability, impacted by:

- Your recovery strategy
- Sharing of state and data
Trade-offs

Scalability

Consistency

exactly once

at least once

at most once

Recovery Strategy

Availability

Sharing Data

share everything

share something

share nothing

share nothing
- CAPACITY PLANNING -

Capacity planning is the design phase which guarantees that your system can withstand the load it was built to handle, and with time, scaling to handle increased demand.

- No single point of failure
- Cluster blueprint for scalability
- Load Regulation
- Back Pressure
When you start stress testing your systems, you need to look for bottle necks in the:

- Concurrency Model
- I/O
- OS
- NETWORK
Stress Testing
Capacity Testing

- Soak testing
- Spike testing
- Stress testing
- Load testing
STEPS AIMED AT **SCALABILITY**

7. Design your cluster blueprint, looking at node ratios for scaling up and down.
8. Identify where to apply back-pressure and load regulation.
Monitoring & Preemptive Support

Through monitoring and pre-emptive support, the five nines once reserved for Telecom systems are now easily attainable in any other vertical.
If you do not have detailed snapshots of the system, troubleshooting will be not be methodical and you will have to rely on guesswork. Collect information using:

- Logs
- Alarms
- Metrics
- Monitoring and Pre-emptive Support -

Logs could include:

- SASL Logs
- Alarm Logs
- State changes in the business logic
- System state changes
MONITORING AND PRE-EMPTIVE SUPPORT

Metrics could include

- System Metrics
- Business Metrics
- **MONITORING AND PRE-EMPTIVE SUPPORT** -

Alarms could include

- Metrics based Alarms
- System Alarms
- Business Logic specific Alarms
- **MONITORING AND PRE-EMPTIVE SUPPORT** -

Proactive support automation

Preemptive support automation
- OPERATIONS, DEVOPS AND MONITORING -

9. Define your OAM approach, defining system and business alarms, logs, and metrics.

10. Identify where to apply support automation.
1. Split up your system's functionality into manageable, stand-alone nodes.
2. Decide what distributed architectural pattern you are going to use.
3. Decide what network protocols your nodes, node families and clusters will use when communicating with each other.
4. Define your node interfaces, state and data model.
5. Split up your system's functionality into manageable, stand-alone nodes.
6. For all your data and state, pick your sharing strategy across node families, clusters and types, taking into consideration the needs of
5. Split up your system’s functionality into manageable, stand-alone nodes.

6. For all your data and state, pick your sharing strategy across node families, clusters and types, taking into consideration the needs of your retry strategy.

7. Design your cluster blueprint, looking at node ratios for scaling up and down.

8. Identify where to apply back-pressure and load regulation.

9. Define your O&M approach, defining system and business alarms, logs and metrics.

10. Identify where to apply support automation.
- BIG IDEAS -

- Build systems with no single point of failure.
- Understand the limits of consistency and reliability when scaling your system.
- Understand the tradeoffs you need to make when designing your system.
- Don’t under estimate the importance of capacity planning, monitoring and preemptive support.
THANK YOU!
Any questions?

francesco@erlang-solutions.com
www.erlang-solutions.com
@francescoC