The Design of Scalable Distributed (SD) Erlang

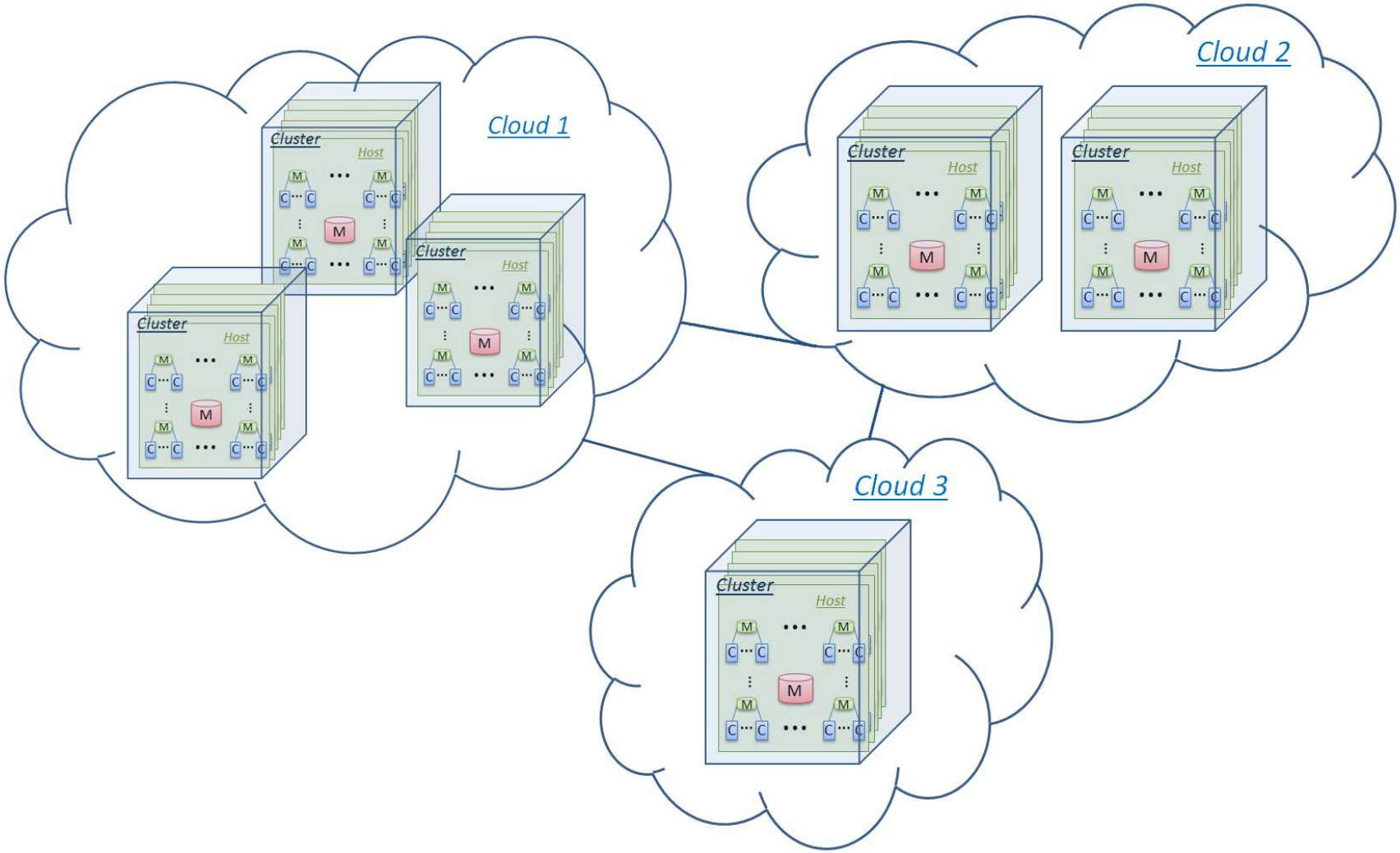
Natalia Chechina, Amir Ghaffari, Phil Trinder, and RELEASE team

June 15, 2012

Design Approach

- Typical hardware architecture
- Scaling
  - Persistent data structures
  - In-memory data structures
  - Computation
- Security

Typical architecture – 10^5 cores

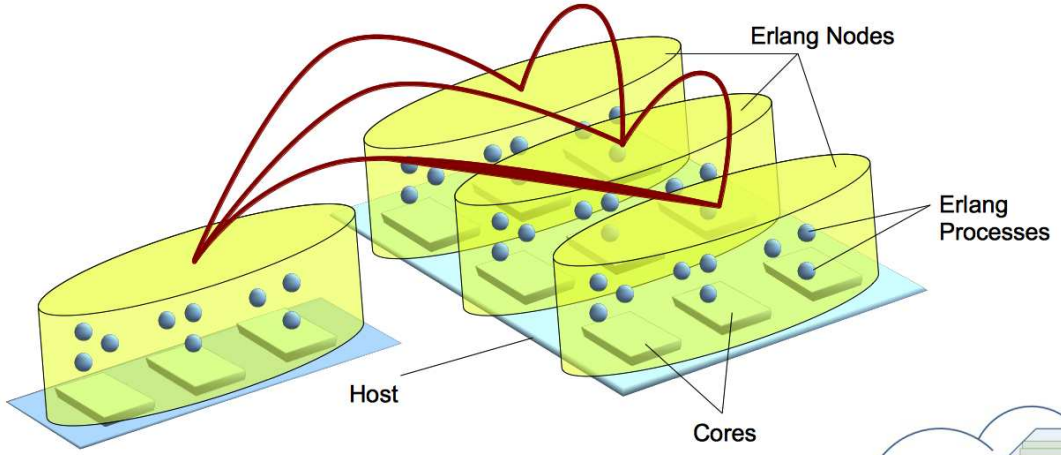


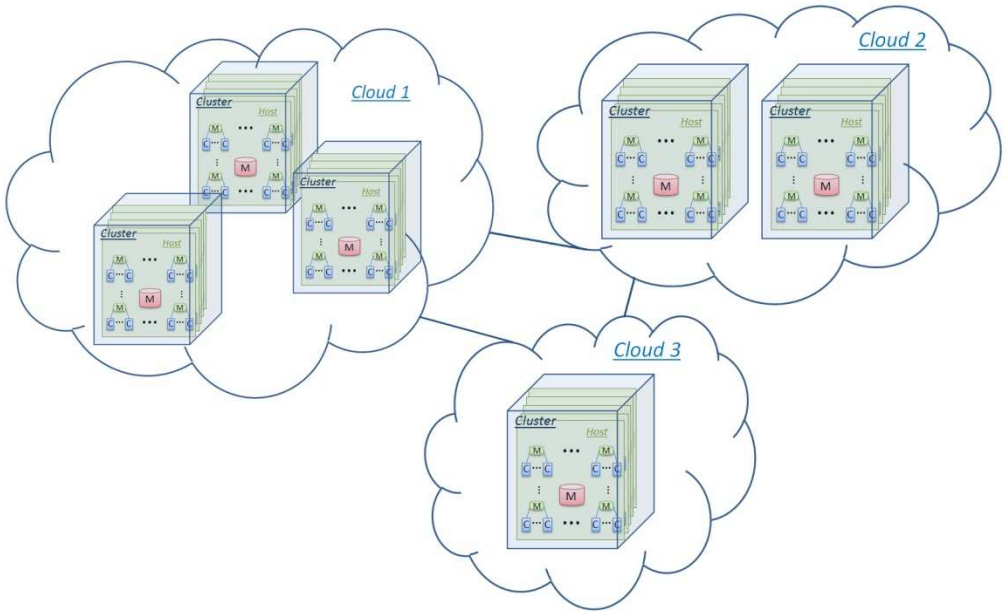
RELEASE

Design Approach

- Typical hardware architecture
- Scaling
  - Persistent data structures
  - In-memory data structures
  - Computation
- Security

Distributed Erlang





RELEASE

Scaling Computation

- Network Scalability
  - All to all connections are not scalable onto 1000s of nodes
  - Aim: Reduce connectivity
- Semi-explicit Placement
  - Becomes not feasible for a programmer to be aware of all nodes
  - Aim: Automatic process placement in groups of nodes

Design Principles

General:

- Working at Erlang level as far as possible
- Preserving the Erlang philosophy and programming idioms
- Minimal design changes

Design Principles

Reliable Scalability:

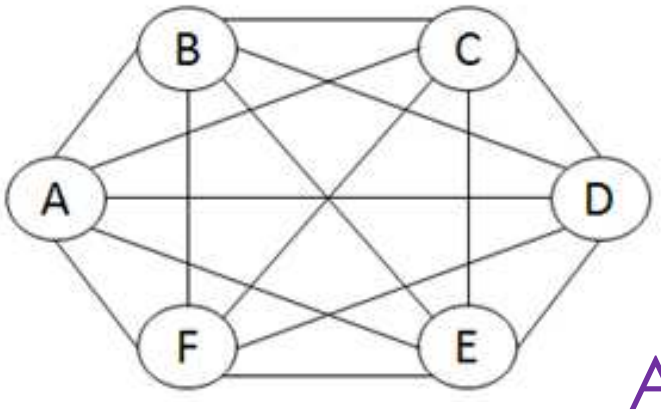
- Avoiding global sharing
- Avoiding explicit prescription
- Introducing an abstract notion of communication architecture
- Keeping Erlang reliability model unchanged as far as possible

Network Scalability

- Grouping nodes in Scalable groups (s_groups)
  - **transitive** connections with nodes of the same s_group
  - **non-transitive** connections with other nodes
- Types of s_groups:
  - Hierarchical
  - Overlapping
  - Partition
- Using **s_group** variables instead of **global** variables: *Var@Group*

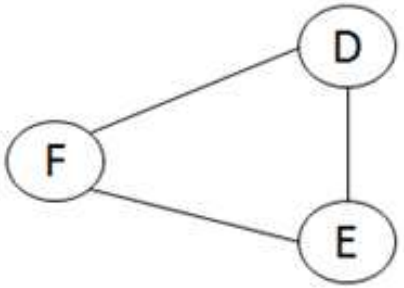
RELEASE

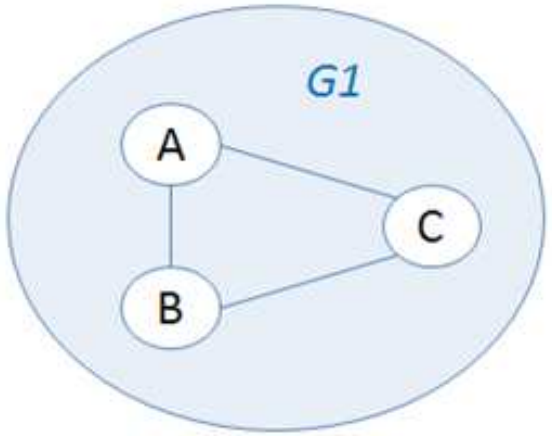
Creating an s_group



a)

A: new_s_group(G1, [A, B, C]).



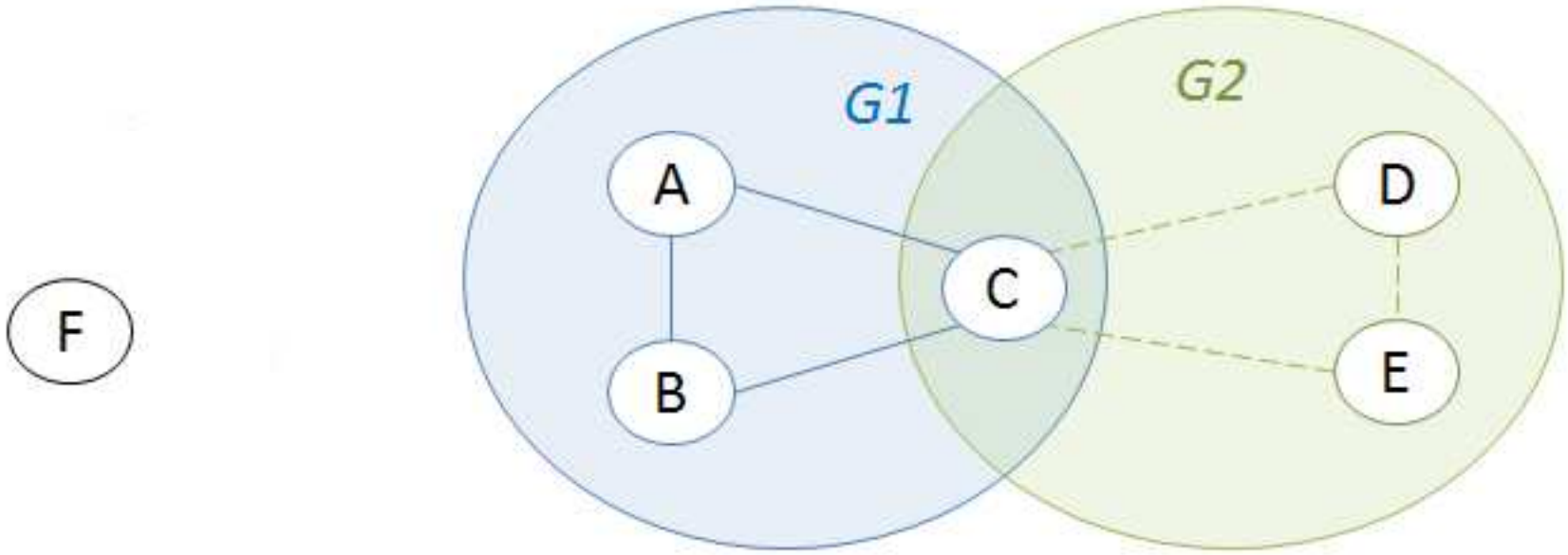


b)

RELEASE

Overlapping Groups & Non-transitive Connections

C: new_s_group(G2, [C, D, E]).

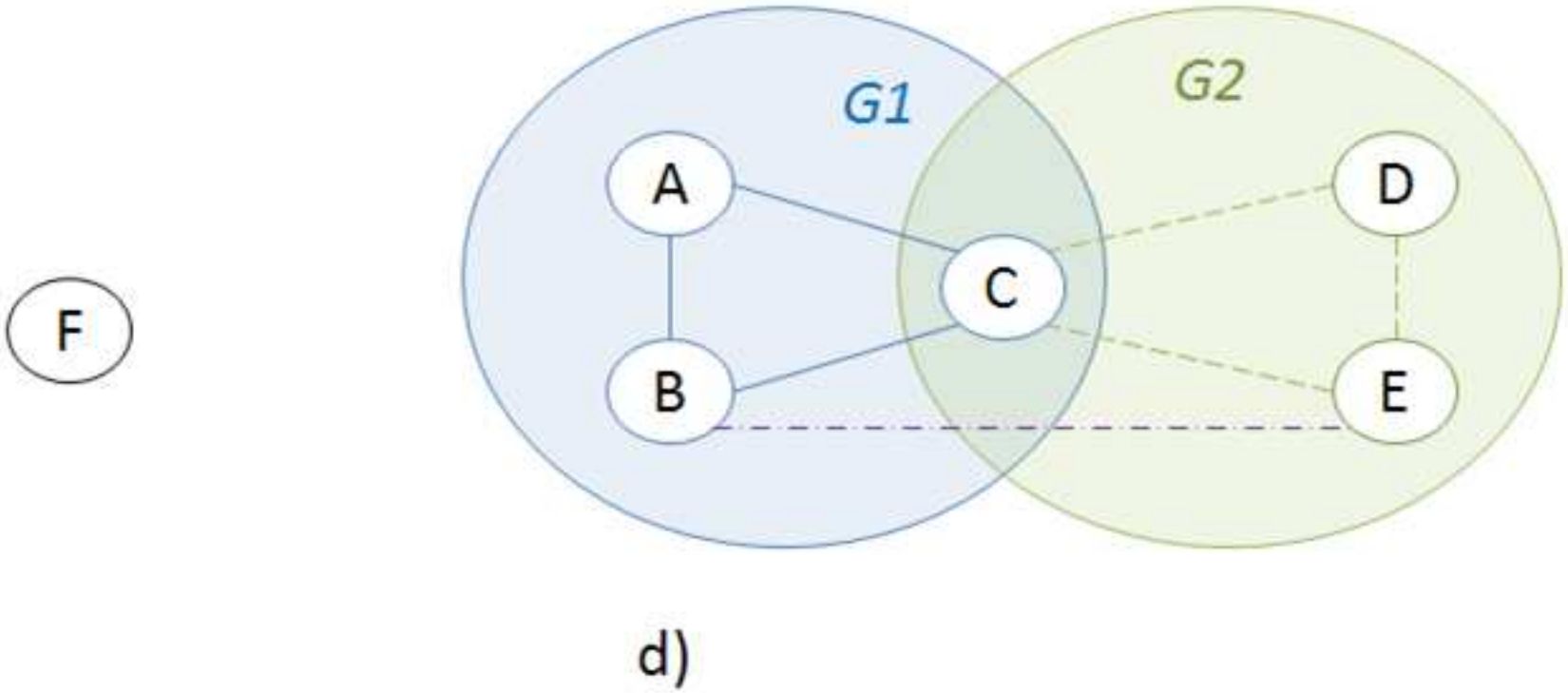


c)

RELEASE

Any to Any Connection

B: spawn(E, *f*).



d)

RELEASE

s_group Functions (1)

- Creating a new s_group

  new_s_group(S_GroupName, [Node]) -> true | *{error, ErrorMsg}*

- Deleting an s_group

  del_s_group(S_GroupName) -> true | *{error, ErrorMsg}*

- Adding new nodes to an existing s_group

  add_node_s_group(S_GroupName, [Node]) -> true | *{error,* ErrorMsg*}*

- Removing nodes from an existing s_group

  remove_node_s_group(S GroupName, [Node]) -> true | *{error,* ErrorMsg*}*

RELEASE

s_group Functions (2)

- Monitoring all nodes of an s_group

  monitor_s_group(S_GroupName) -> ok | *{error, ErrorMsg}*

- Sending a message to all nodes of an s_group

  send_s_group(S_GroupName, Msg) -> Pid | *{badarg, Msg}* *{error, ErrorMsg}*

- Listing nodes of a particular s_group

  s_group_nodes(S_GroupName) -> [Node] | *{error, ErrorMsg}*

- Listing s_groups that a particular node belongs to

  node_s_group_info(Node) -> [S_GroupName]

RELEASE

Scaling Computation

- Semi-explicit Placement
  - Becomes not feasible for a programmer to be aware of all nodes and place each of them explicitly
  - Aim: Automatic process placement

chose_node/1

chose_node(Restrictions) -> node()

Restrictions = [Restriction]

Restriction = *{s_group, S_Group}*

| *{min_dist, MinDist :: integer() >= 0}*

| *{max_dist, MaxDist :: integer() >= 0}*

| *{ideal_dist, IdealDist :: integer() >= 0}*

```
start() ->
        TargetNode = chose_node({s_group, S_Group},
        {ideal_dist, IdealDist}),

        spawn(TargetNode, fun() -> loop() end).
```

RELEASE

Thank you!