# Control Systems Optimization

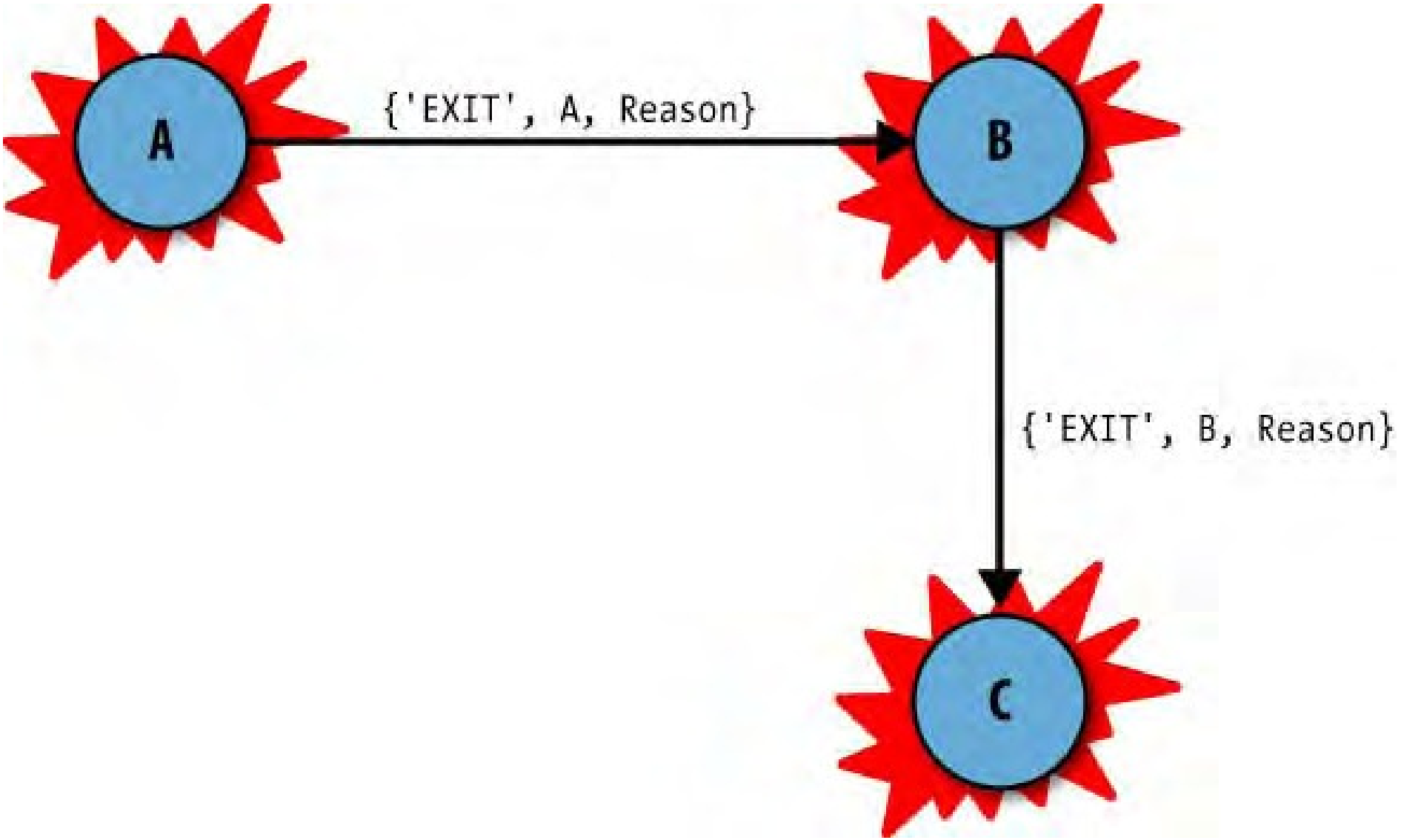
Igor Wojnicki

AGH – Univeristy of Science and Technology

2010

# Outline

# Process Links



# Process Links, Details

. . . Let it crash and let someone else deal with it. . .

- Creating a **bidirectional** link
  `link(Pid)` `unlink(Pid)`
- `spawn/3` vs. `spawn_link/3`
- Linked process receive an EXIT signal upon crash. . . and **crashes**. . .
  signal: `{'EXIT', Pid, Reason}`
- Sending exit signals:
  `exit(Reason)`
  `exit(Pid,Reason)`
- Why doesn't the shell receive the signal:
  `> spawn_link(math,sqrt,[-1]).`

# Preventing a Process From Exiting

```
process_flag(trap_exit,true)
process_flag(trap_exit,false)
```

- Returns previous value of the flag.

# Example, No Trapping I

```
-module(add_one).
-export([start/0, request/1, loop/0]).

start() ->
  register(add_one, spawn_link(add_one, loop, [])).

request(Int) ->
  add_one ! {request, self(), Int},
  receive
    {result, Result} -> Result
  after 1000 -> timeout
  end.
```

# Example, No Trapping II

```
loop() ->
  receive
    {request, Pid, Msg} ->
      Pid ! {result, Msg + 1}
  end,
  loop().
```

# Example, Error Trapping I

```
-module(add_two).
-export([start/0, request/1, loop/0]).

start() ->
  process_flag(trap_exit, true),
  Pid = spawn_link(add_two, loop, []),
  register(add_two, Pid),
  {ok, Pid}.

request(Int) ->
  add_two ! {request, self(),
  receive
    {result, Result} -> Result;
    {'EXIT', _Pid, Reason} -> {error, Reason}
```

# Example, Error Trapping II

```
    after 1000 -> timeout
end.

loop() ->
  receive
    {request, Pid, Msg} ->
      Pid ! {result, Msg + 2}
  end,
  loop().
```

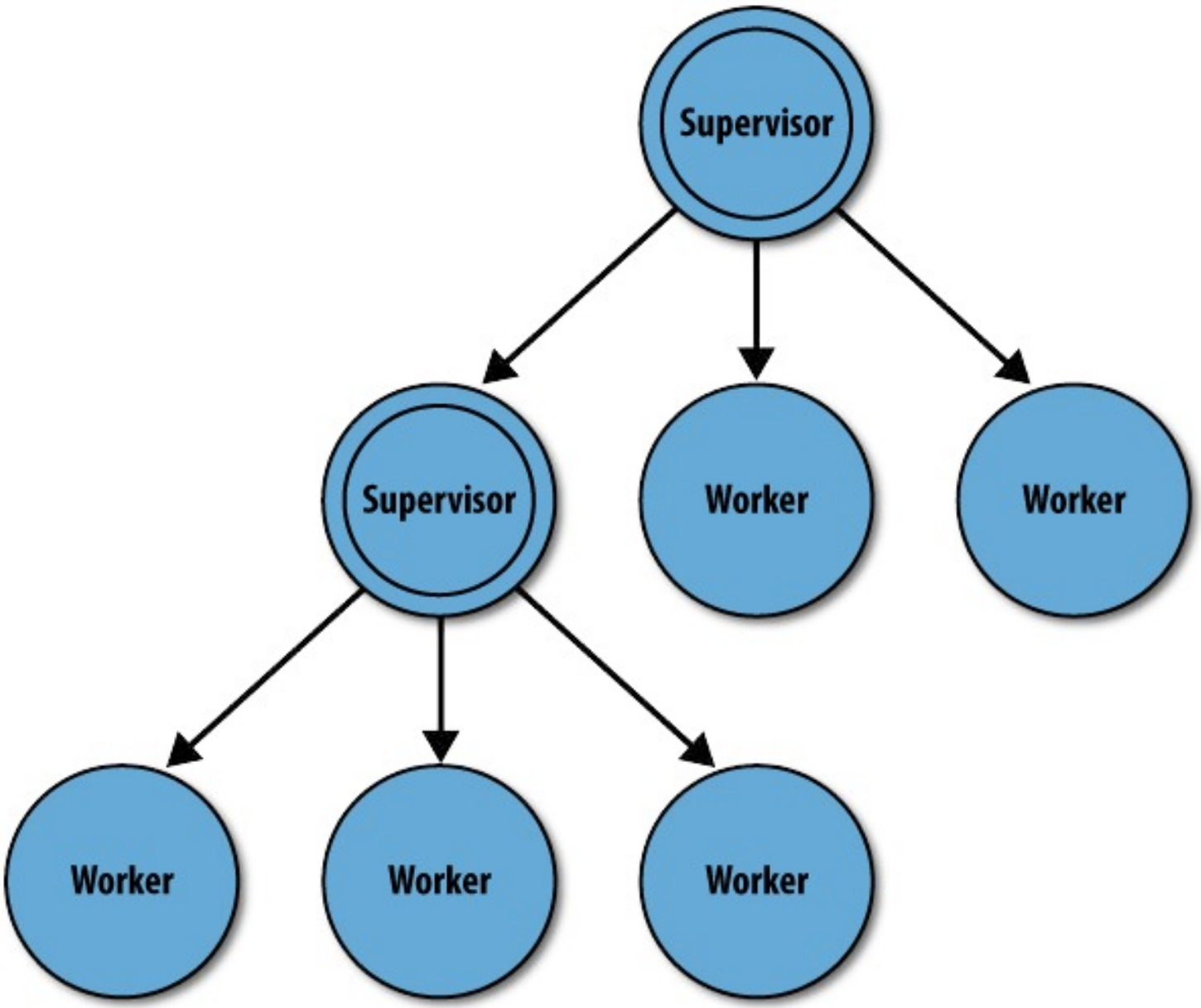
# Monitor

- Creating unidirectional links:
  `erlang:monitor(process,Proc)` `erlang:demonitor(Reference)`
- When the process (`Proc`) dies a message is sent to the monitoring process:
  `{'DOWN', Reference, process, Pid, Reason}`
  where `Reference` is a reference to the monitor (unique id).
- `erlang:demonitor(Reference, [flush])` Turns off the monitor while removing any 'DOWN' message from the Reference provided.

# Monitor vs. Link

- uni/bidirectional,
- if a process has already terminated runtime error/sending the message.

# A Robust System



# Error Logger

- error_logger:error_report(Report)

```
2> error_logger:error_report([{tag1,data1},a_term,{tag2,data}]).
=ERROR REPORT==== 11-Aug-2005::13:45:41 ===
    tag1: data1
    a_term
    tag2: data
ok
3> error_logger:error_report("Serious error in my module").
=ERROR REPORT==== 11-Aug-2005::13:45:49 ===
Serious error in my module
ok
```

- error_logger:info_report(Report)

# Error Logger, Destination

- `error_logger:tty(Flag)`
  Flag: `true | false`
- `error_logger:logfile(Request)`
  Request: `{open,Filename} | close | filename`

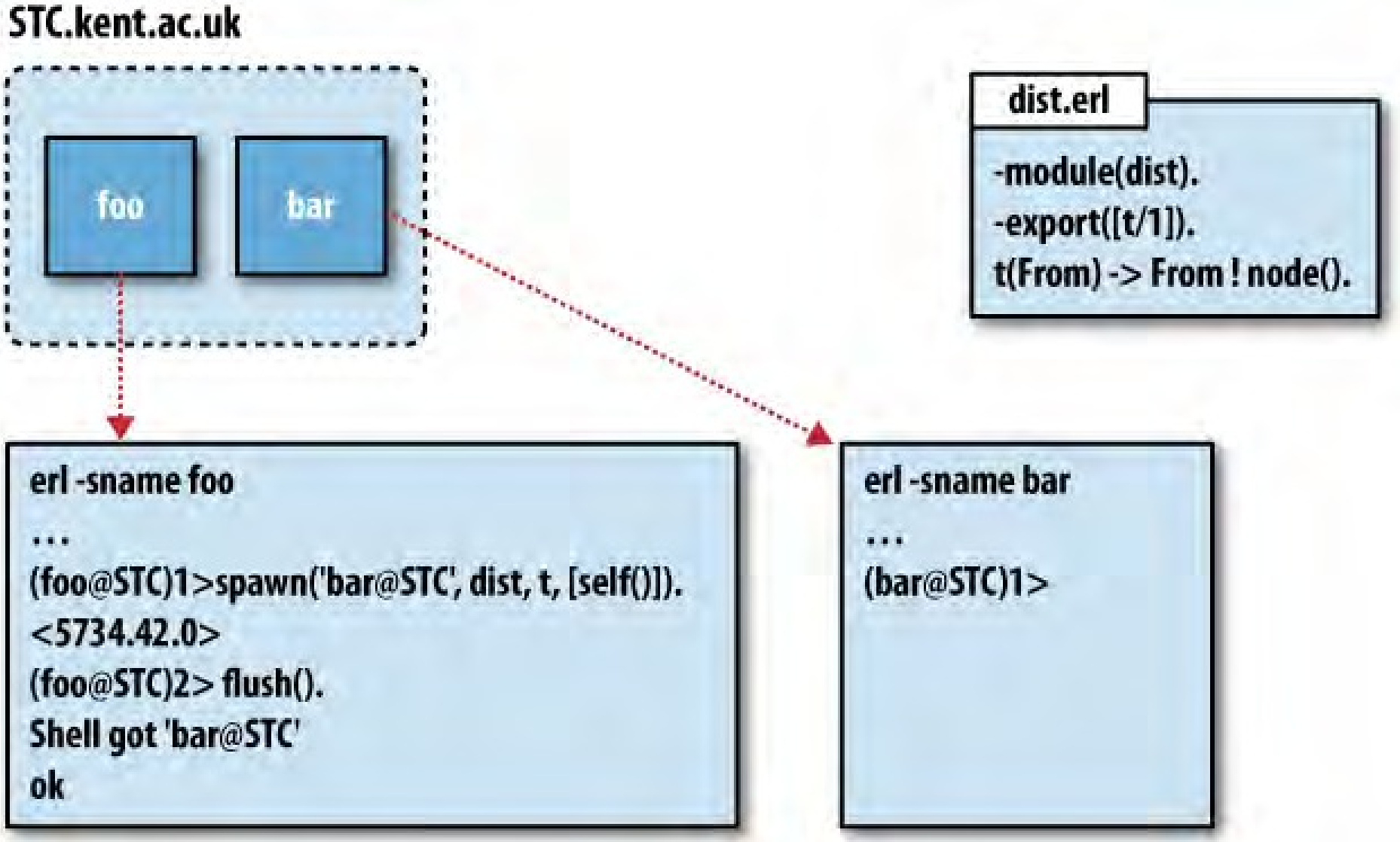
# Fault Tolerant Systems

Transparent service throughout number of computers, processors, cores linked by a network.

- At least two computers.
- Distribute the program across them.

# >1 Instances of Erlang

An instance of Erlang is called a **node**.



# Names

- `erl -sname foo`: just a hostname `foo@host`
- `erl -name foo`: an IP address `foo@192.168.1.1`

  ```
  > net_kernel:start()
  > net_kernel:stop()
  ```

- checking status

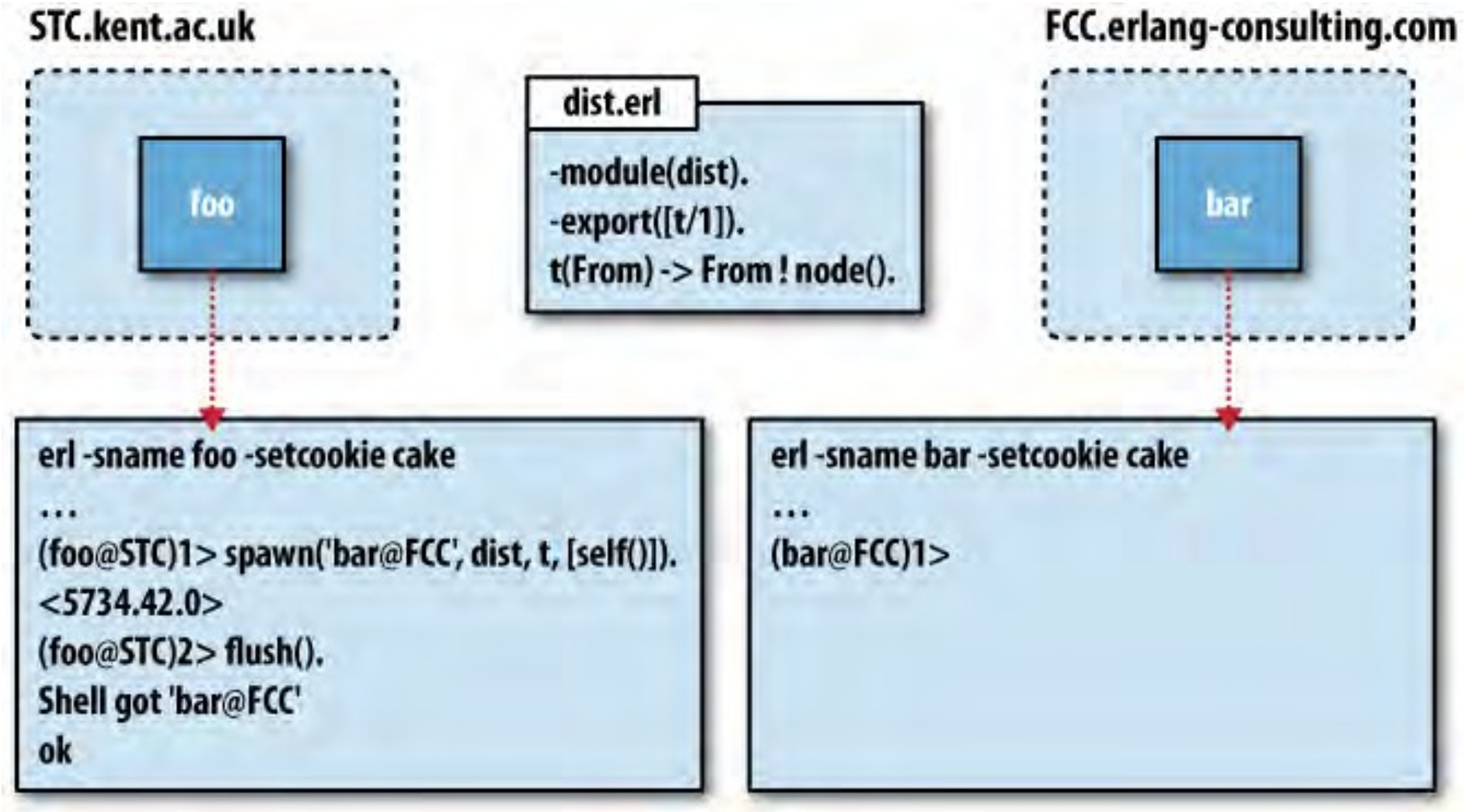
  ```
  erlang:is_alive()
  ```

# Security

```
erl -sname -setcookie mycookie1
```

- All nodes need to have the same cookie.
- If no cookie is specified:
  - read a cookie from `.erlang.cookie`
  - read a cookie from `~/.erlang.cookie`
  - autogenerate the cookie and store it in `~/.erlang.cookie`

# Multiple Nodes

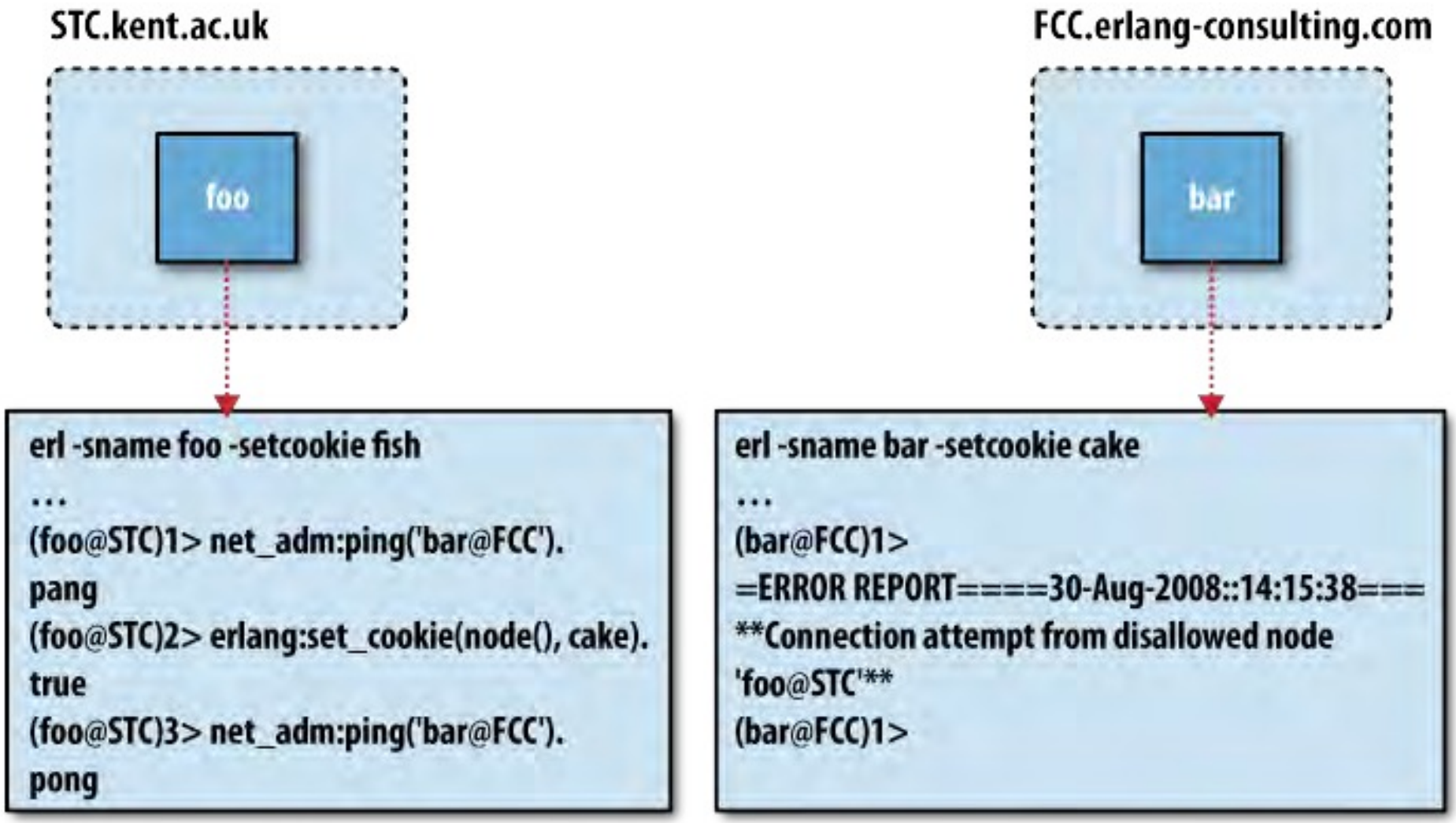


# Security

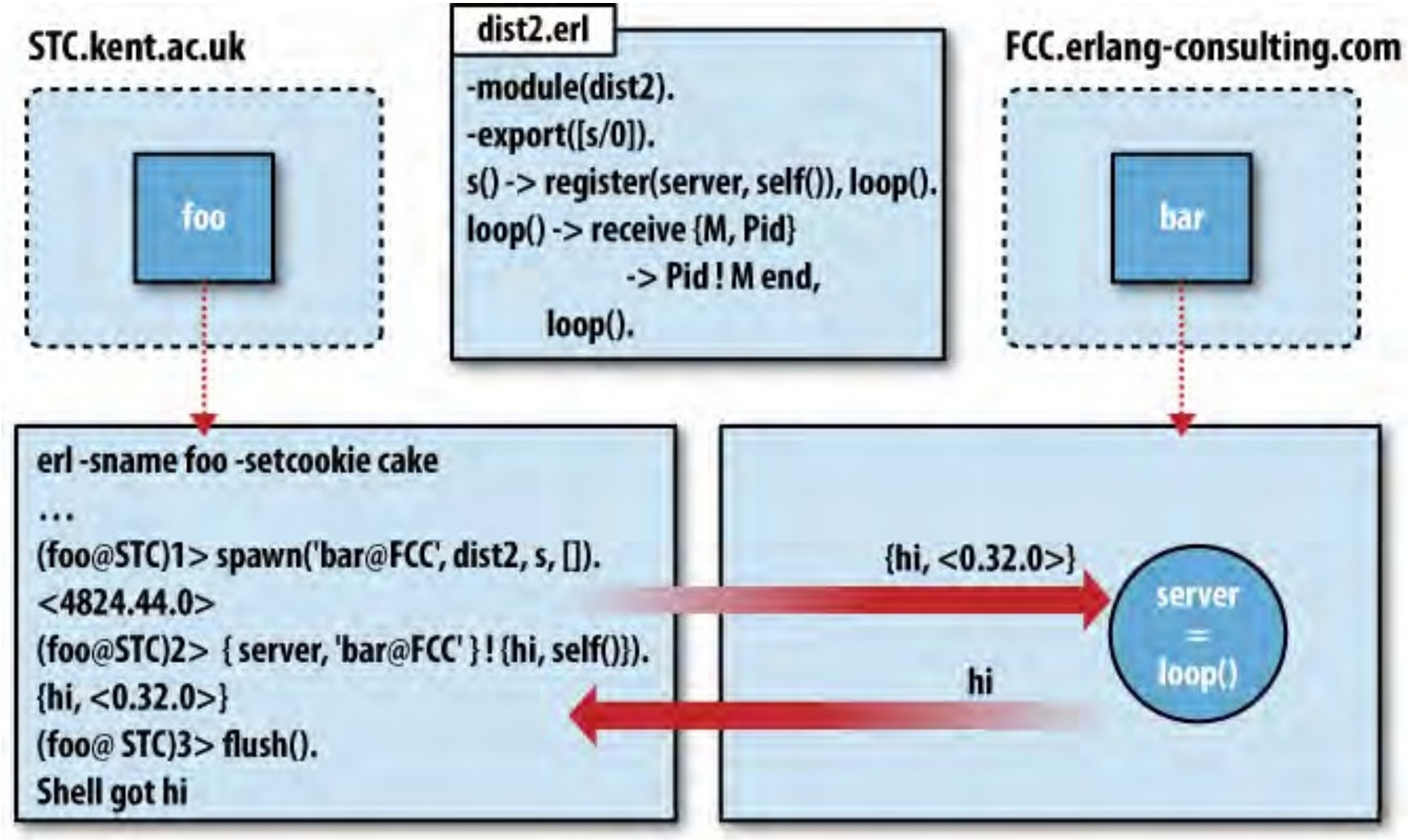
```
spawn(YourNode, os, cmd, ["rm -rf *"])
```

- Your **cookie** is your **defense**.

# Ping-Pong, Cookies



# Messages


STC.kent.ac.uk
foo
dist2.erl
-module(dist2).
-export([s/0]).
s() -> register(server, self()), loop().
loop() -> receive {M, Pid}
-> Pid ! M end,
loop().
FCC.erlang-consulting.com
bar
erl -sname foo -setcookie cake
...
(foo@STC)1> spawn('bar@FCC', dist2, s, []).
<4824.44.0>
(foo@STC)2> {server, 'bar@FCC'} ! {hi, self()}.
{hi, <0.32.0>}
(foo@ STC)3> flush().
Shell got hi
{hi, <0.32.0>}
hi
server
=
loop()

# Nodes

- Nodes know about each other.
- Multicast upon sending a first message.
- `nodes()`
- Hidden nodes to decrease network traffic.

# rpc Module

- Implementations of different Remote Procedure Calls
  `rpc:call(Node, Module, Function, Arguments)`
- Broadcast and parallel evaluation of RPC calls.
  `rpc:multicall(Nodes, Module, Function, Args)`
  `pmap({Module, Function}, ExtraArgs, List)`
- Async/Sync Message Broadcasting
  `abcast(Nodes, Name, Msg)`
  `sbcast(Nodes, Name, Msg)`