Experiments in OTP-Compliant Dataflow Programming

Introducing Erlang Services Platform (Erlang/SP)

San Francisco Erlang Factory, March 21, 2013 *Jay Nelson* *Twitter: @duomark* *Email: Jay@duomark.com*

Manycore Concurrency

- Erlang/OTP encourages server-style programming
    - One process with a potentially large internal state
    - Serialized mutations to the internal state
    - No easy way to automatically break up a gen_* process
- Modern CPUs will soon have 100, 1K or even 10K cores
- How can Erlang programmers adapt to this future?

Keeping Cores Busy

- OS-Level virtualization (multiply your problem)
  - One CPU appears to be 100s of machines
  - Many tenants and applications run on the same hardware
- Single application concurrency (divide and conquer)
  - Requires many fine-grained tasks
  - Implies that existing state must be distributed to more processes

Erlang/SP

- OTP-compliant library
  - Open source at `https://github.com/duomark/erlangsp`
  - Can be included directly with `rebar.config`
  - Undergoing active development and evolution
- Encourages the use of "services" over "servers"
  - Service: set of co-ops implementing an independent subsystem
  - Co-op: tightly bound graph of cooperating processes

Example Services for Texting

- Presence - users, bots or services that are online
- Connection listener - accepts user client connections
- Message routing - delivers messages from one user to others
- Attachment management - stores and ids attachments (image, sound)
- Push notifications - message count badges sent to offline users
- User search - discover users of the texting application

Goals of Erlang/SP

- Simplify and encourage the creation of massive concurrency
  - Automate the generation of process networks
  - Map mutable state to a structural representation of all states
  - Use data flow to stimulate the network maps
- Allow incremental integration with existing OTP code
- Provide tools for understanding high concurrency performance

Process Networks

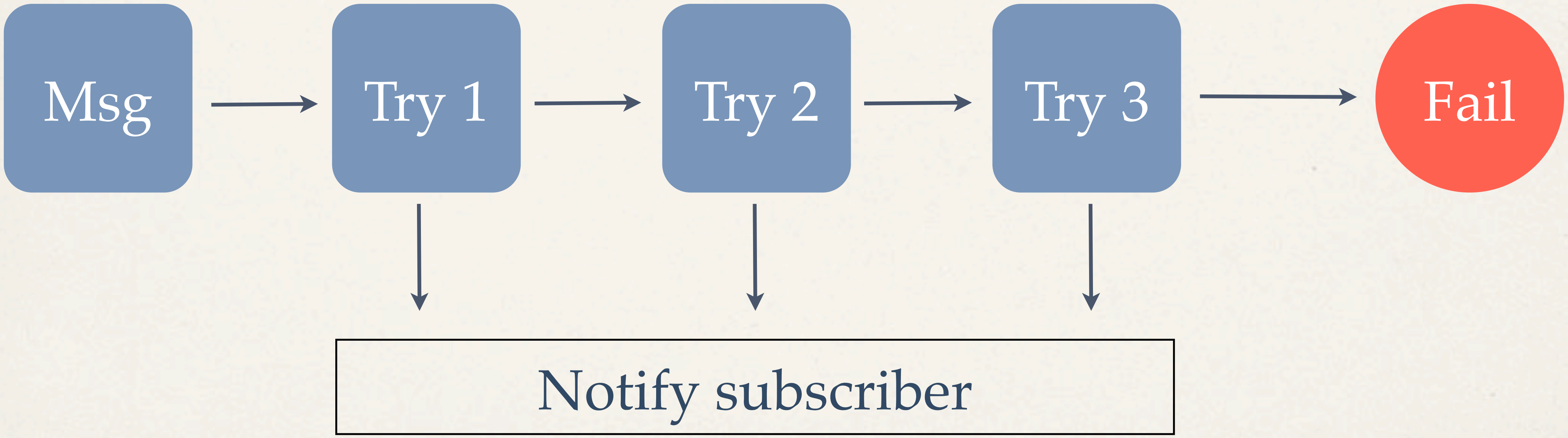
- Collection of processes wired together along messaging lines
  - Build a Directed Acyclic Graph (DAG) template
  - Create a co-op with a task function process per graph node
  - Each process knows only its downstream receivers
  - Inject data to propagate through the computation network
- Glue networks together with a mixture of OTP and SP constructs

Networked State Representation

- Each path through a network is unique
- Each path equals state
    - Arrival at a specific node implies the path taken
    - Process at that node has implicit state knowledge
- The process network embodies all states reachable
- Tradeoff: mutable state vs. processes + messaging
    - Adding latency, but increasing concurrency

Networked State (cont.)

Replacing a counter with a pipeline of processes



Msg

Try 1

Try 2

Try 3

Fail

Notify subscriber

Networked State (cont.)

- Programming with Erlang/SP
  - The art of disassembling and distributing state
  - Selecting network patterns that describe the problem space
- Functional decomposition
  - Choosing the smallest meaningful function granularity
  - Mapping functions to separate processes

OTP-Compliance

- Processes which:
    - respond to system messages (*dbg*, *trace*, etc)
    - can be supervised (deal with *'EXIT'* messages properly)
    - reply to reltool *get_modules* request
- Compatible with all OTP tools
- Can integrate freely with OTP constructs (e.g., *gen_server*, *supervisor*)
- Support software upgrade in the context of a larger OTP system

Example message loop code (erlangsp: *coop_head_ctl_rcv.erl*)

```
msg_loop({} = State, Root_Pid, Timeout, Debug_Opts) ->

    receive

        %% System messages for compatibility with OTP...
        {'EXIT', _Parent, Reason}  -> exit(Reason);

        {system, From, System_Msg} ->
            Sys_Args = {State, Root_Pid, Timeout, Debug_Opts},
            handle_sys(Sys_Args, From, System_Msg);

        {get_modules, From}        ->
            From ! {modules, [?MODULE]},
            ?MSG_LOOP_RECURSE;

        ?CTL_MSG({init_state, #coop_head_state{} = New_State}) ->
            ?MSG_LOOP_RECURSE(New_State)
    end;
```

OTP-Compliance (cont.)

- More details in my Vancouver 2012 Erlang Factory Lite talk
  - Managing Processes without OTP (and how to make them OTP-compliant)
  - http://www.erlang-factory.com/upload/presentations/674/OTPProcs.pdf

Implementation Details

1. Esp_service behaviour
2. Esp_tcp_service behaviour
3. Esp_epmd

Esp_service Behaviour

- Erlang/SP library provided behaviour
- Client module must implement
  - `new(Args, Receiver) -> esp_service() | {error, _}.`
  - `start(Service, Proplist) -> esp_service() | {error, _}.`
  - `stop(Service) -> esp_service().`
- Services don't run on creation, until explicitly started
- A collection of services plus admin/control logic make a system

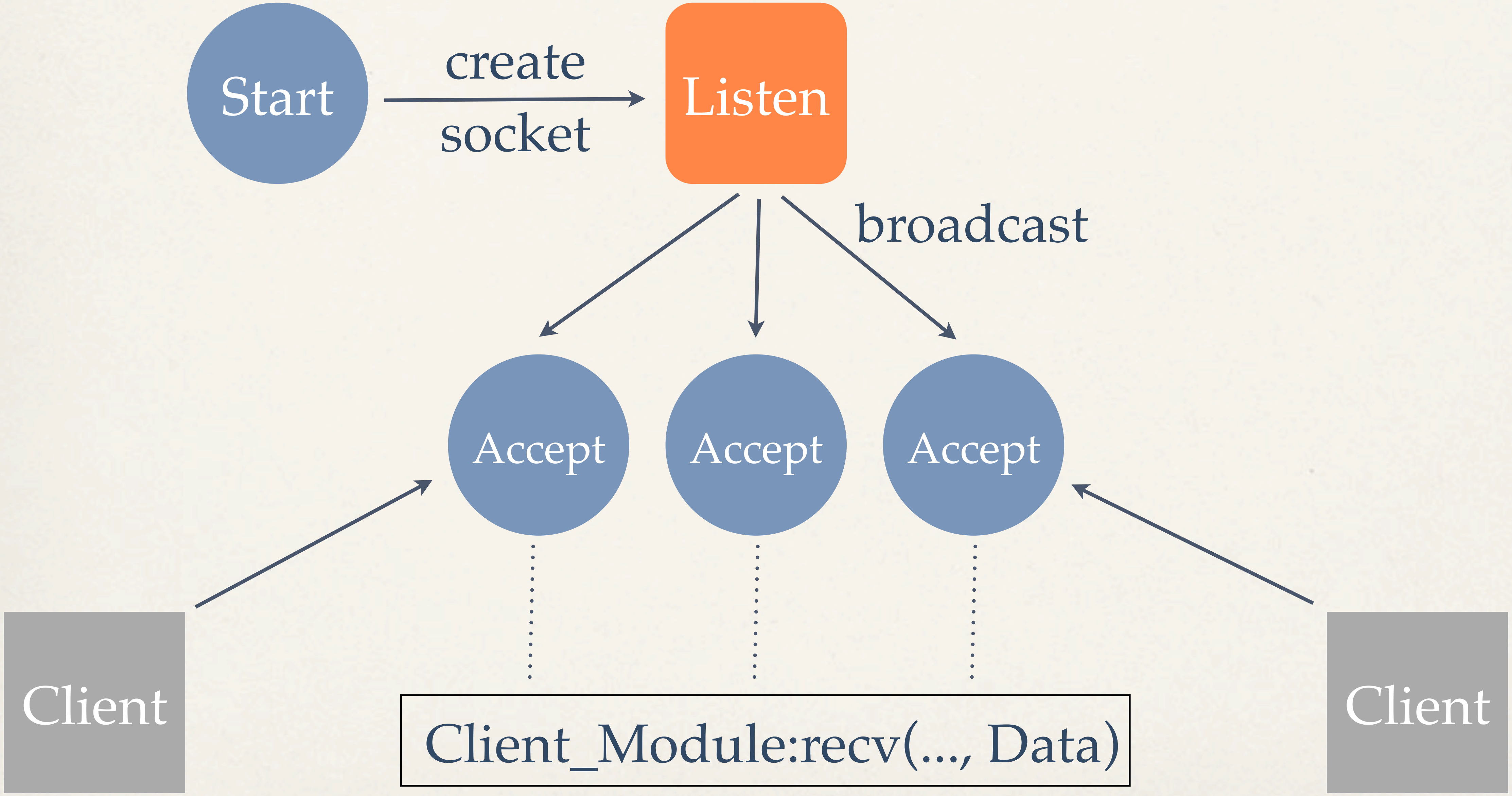
Esp_service Behaviour (cont.)

- Built-in functions
  - `make_service(coop()) -> esp_service().`
  - `link_service(esp_service()) -> ok.`
  - `status(esp_service()) -> svc_state().`
  - `act_on(esp_service(), Data) -> ok | {error, not_started}`
  - `suspend, suspend_for / resume, resume_after`
  - `set_overload / is_overloaded`

Esp_tcp_service Behaviour

- Generic service for accepting TCP connections (like ranch or swarm)
  - Uses prim_inet:async_accept internally
  - Implements esp_service interface using a fanout co-op graph
  - Client module handles incoming data
  - Client module can be changed after acceptor is launched
- Listen socket, plus acceptors are all linked
  - Client module can remove links on connect or data recv

Diagram of Esp_tcp_service



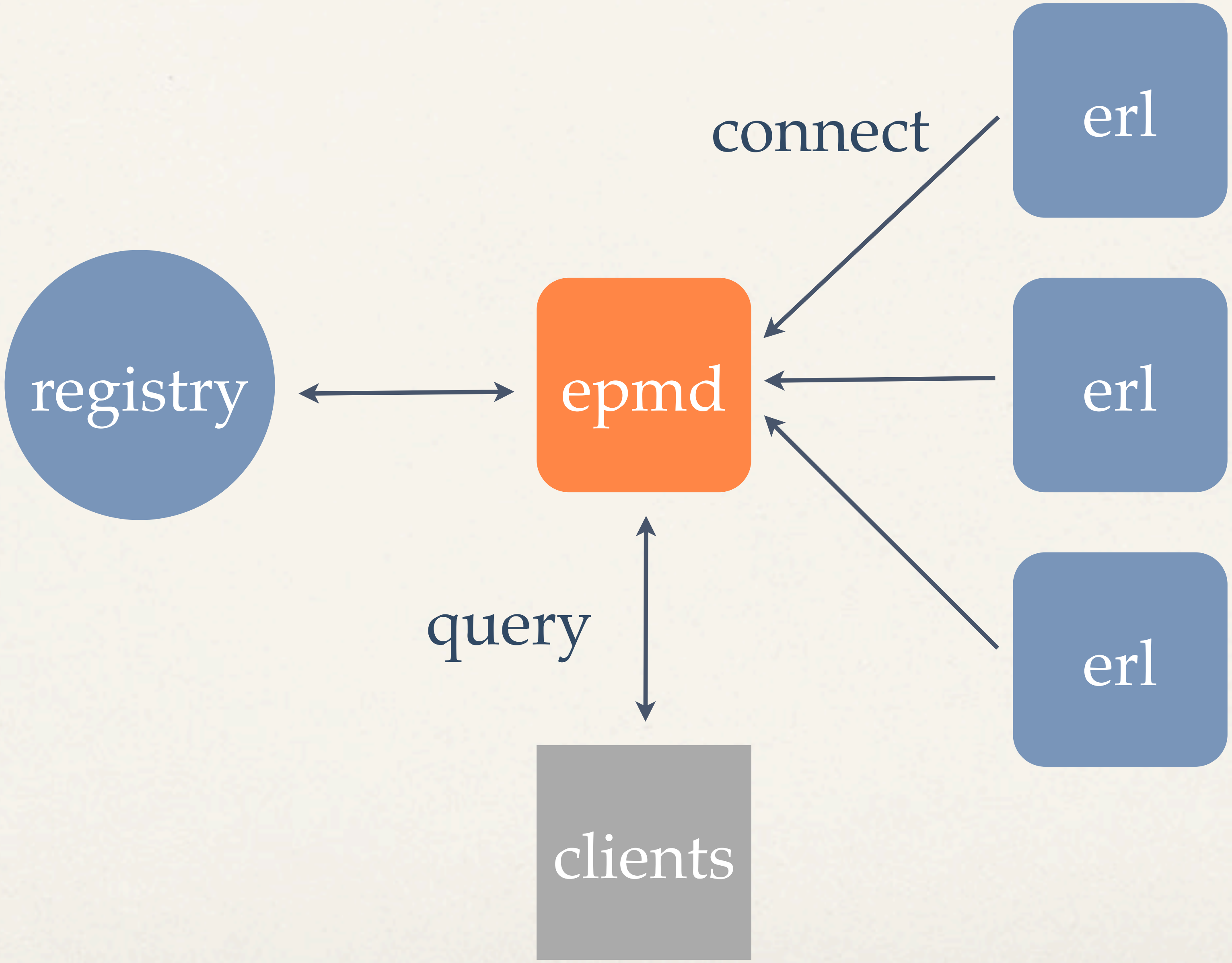
Esp_tcp_service Behaviour (cont.)

- Fanout broadcast from socket listen to N acceptor children
    - No downstream receivers from fanout
    - Connection is kept in acceptor process, no tcp socket transfer
    - Client module is free to generate side effects on data recv
    - On completion, acceptor process is removed from co-op
    - Replacement is slab allocated for higher volume performance

Epmd daemon

- Epmd maintains connection registry
    - List of node name, port for shared cookie Erlang nodes
    - Part of base distribution, written in C
- TCP to local node when erlang VM starts
- Epmd accepts queries for active nodes

Diagram of Epmd



Typical vs. Erlang/SP version

- Typical
    - TCP connection listener pool
    - Ets table of connections
    - Central server for queries
- Erlang/SP style
    - Esp_tcp service (connection listener fanout)
    - Query service (fanout of connected nodes)

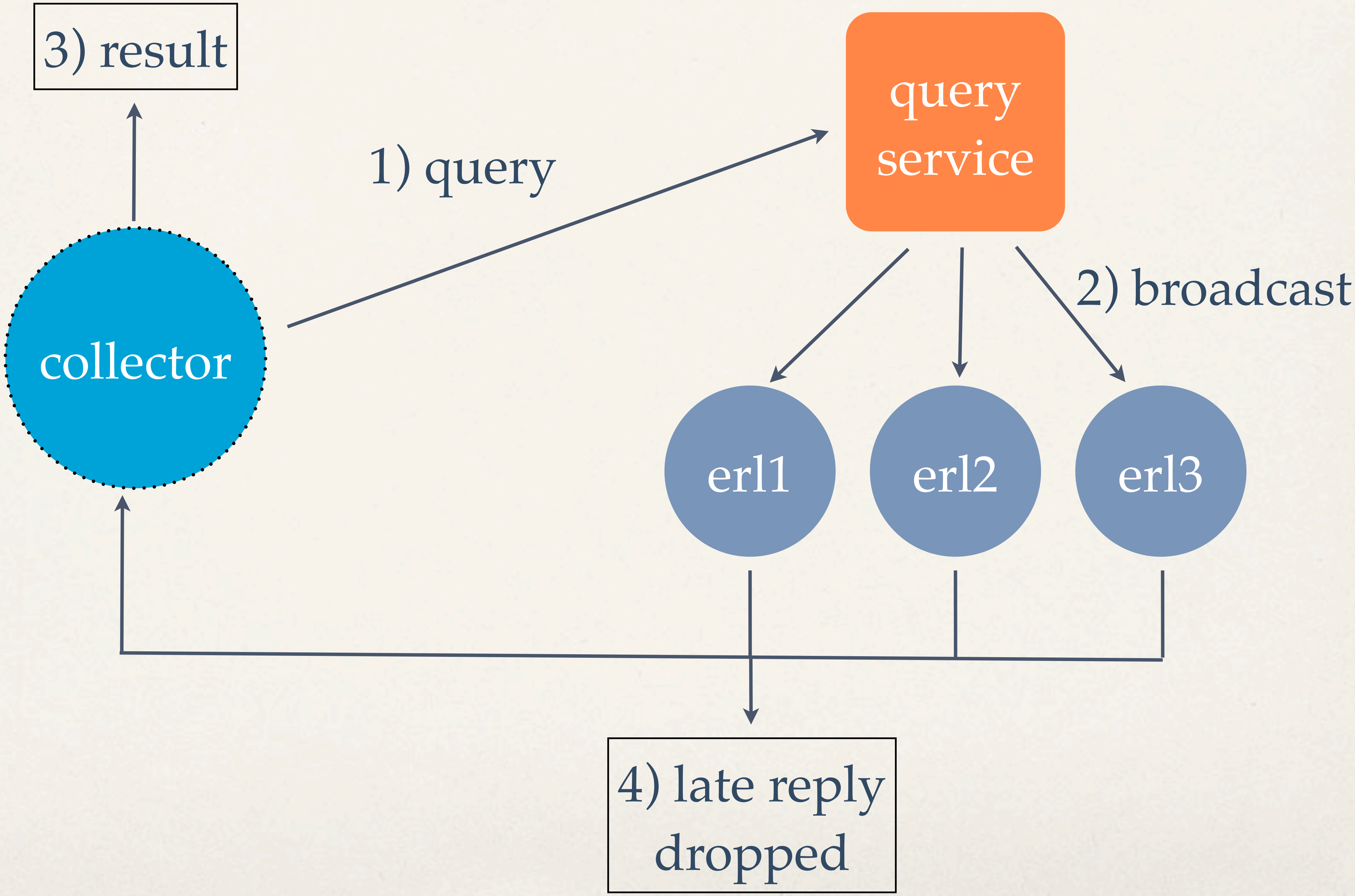
Esp_epmd Service

- Connection fanout is an esp_tcp_service
  - Acceptor task for erl connection is to migrate to query fanout
  - Acceptor task for other requests is reply and die
- Queries are routed to query service fanout
  - Broadcast mode sends query to all live connections
  - Each replies if query matches, ignores if not

Esp_epmd Query Reply

- Asynchronous distribution requires collection of results
- Newly spawned collector task listens for responses
    - Replies must be sent to requestor within timeout
    - Late to arrive messages find no process and are dropped
- After response, collector pid expires

Query Collection



Erlang/SP Contribution

- Trade internal state (ets table) for process graph
    - Database of connections is a fanout graph of processes
    - Query occurs in parallel naturally
    - Entirely eliminates need for mutable state update on connect
- Erlang/SP provides common library patterns
    - Reduction in code to implement epmd logic
    - Full OTP tool set can be used on live epmd connection processes

Conclusion

- Erlang/SP enables higher-level concurrency patterns
  - Eschews state-based, single-server model
  - Supports graph-oriented concurrent algorithm structures
  - Allows integration with existing OTP structures
  - Supports migration of systems to manycore architectures
- v0.1.0 with esp_service behaviours will be announced soon