Real Cloud Computing
Erlang and Elixir User Group
Seattle Washington, June 18th 2014

mjtruog@gmail.com

Cloud computing you can own!

- CloudI is BSD licensed
- CloudI is an interface for developers to provide fine-grained dynamic fault-tolerance across all programming languages
- Non-Erlang programming languages gain fault-tolerance without virtualization (Real!)
- Self-Contained to provide implicit security if deployed privately (everything is open-source)
  → No encryption is completely secure
  → An air gap network is secure

Why does this matter?

- All souce code contains bugs!
  (typically measured as defects per KSLOC [1])
  → Fault-Tolerance matters
- Fault-Tolerance is the main benefit that cloud computing should provide
- Unencumbered by a CLA (Contributor License Agreement), board members, governance, committees or any other impediments to usage

[1] http://www.infoq.com/news/2012/03/Defects-Open-Source-Commercial

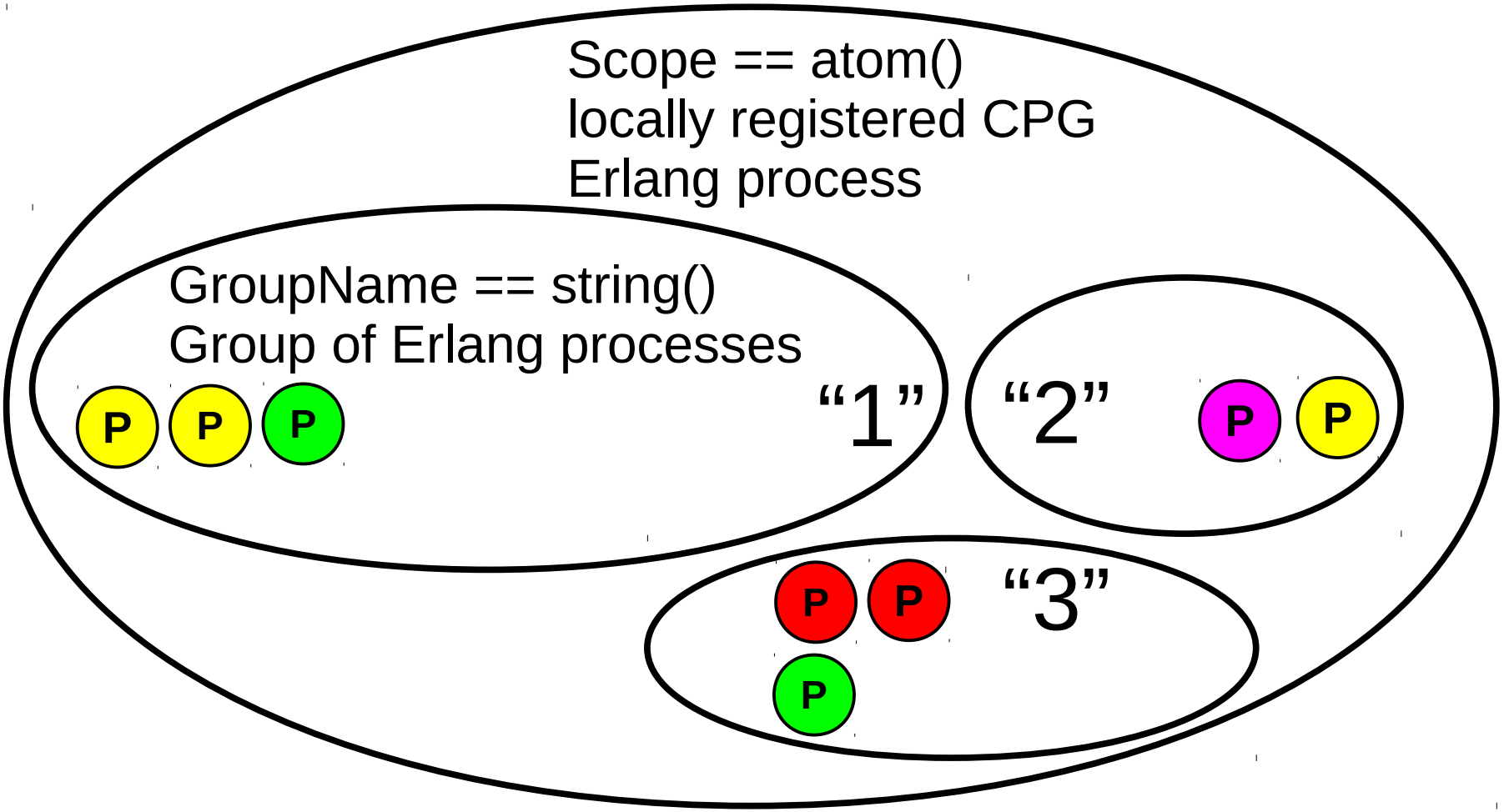
Part 1

CloudI is
Dynamic Fault-Tolerance
for Erlang source code

CPG - The Heart of CloudI

- “CloudI Process Groups” are master-less
- All data is retrieved from the local node but updates are shared with remote nodes
- Based on pg2, includes many improvements (unlike pg2, it works with the via tuple syntax)
- CloudI is AP-type from the CAP theorem (Get Consistency from a database interface!)
- No minimum number of nodes required for CPG to function error-free

CPG - Process Organization



GroupName is called a “Service Name Pattern” within CloudI

CPG - Better Process Pooling

- Poolboy changes internal state to access a pool and queue internally, CPG doesn't
  → CPG is for Flow-Based Programming (FBP)
- Doesn't queue so that queuing can be done with separate granular fault-tolerance (i.e., a CloudI Service)
- CPG [1] is a Conflict-free Replicated Data Type (CRDT)
  → state-based (Convergent) with node monitoring (startup)
  → operation-based (Commutative) with updates (join/leave)
  → provides Strong Eventual Consistency (SEC) [2]
- Handles higher throughput (CPG state caching)
  → No bottleneck on process lookup

[1] https://github.com/okeuday/cpg/
[2] http://dl.acm.org/citation.cfm?id=2050642

CPG - GroupName patterns?

- “*” must consume 1 or more characters
- “**” is forbidden
- “/service/name” matches the patterns: “/service/*”, “/*/name”, “/*/*”, “/*”, “/service/nam*”, etc. [1]
- A “Service Name” is the GroupName string used for the CPG process lookup (the “Service Name Pattern” is what is stored inside CPG)

[1] http://cloudi.org/faq.html#4_NamePattern

What is CloudI?

CloudI provides a service abstraction (running long-lived processes) for many reasons:

- The service abstraction enforces fault-tolerance constraints for all services, in the same way:
  → Timeout, automatically decremented
  → MaxR/MaxT, same as a supervisor
- Encapsulates CPG usage for service name lookups to avoid implementation errors
  → Adds ACLs, service name match on sends
- A service is more dynamic than a gen_server
  → refers to more than 1 service process normally
  → each service name pattern has redundancy

CloudI Scalability Highlights

- count_process_dynamic
  → Rate-based service process counts
- monkey_latency/monkey_chaos
  → Simulated failures (~ Netflix's SimianArmy)
- queue_limit/priority_default
  → Services can limit their incoming queue size
  → All service requests have a priority
  (defaults to 0, -128 high, 127 low)
- count_process/count_thread (service config)
  → Service instances set their initial concurrency

CloudI Memory Consumption

- request_pid_uses/info_pid_uses
  → Control the frequency of heap GC
- Avoids any difficulties with GC latency not keeping up with binary reference death
- Only uses a single Erlang pid (Dispatcher) until the request_pid or info_pid is required (unless duo_mode is enabled) [1]
- queue_limit limits the queued service requests
  → Erlang pid messages are put into the heap

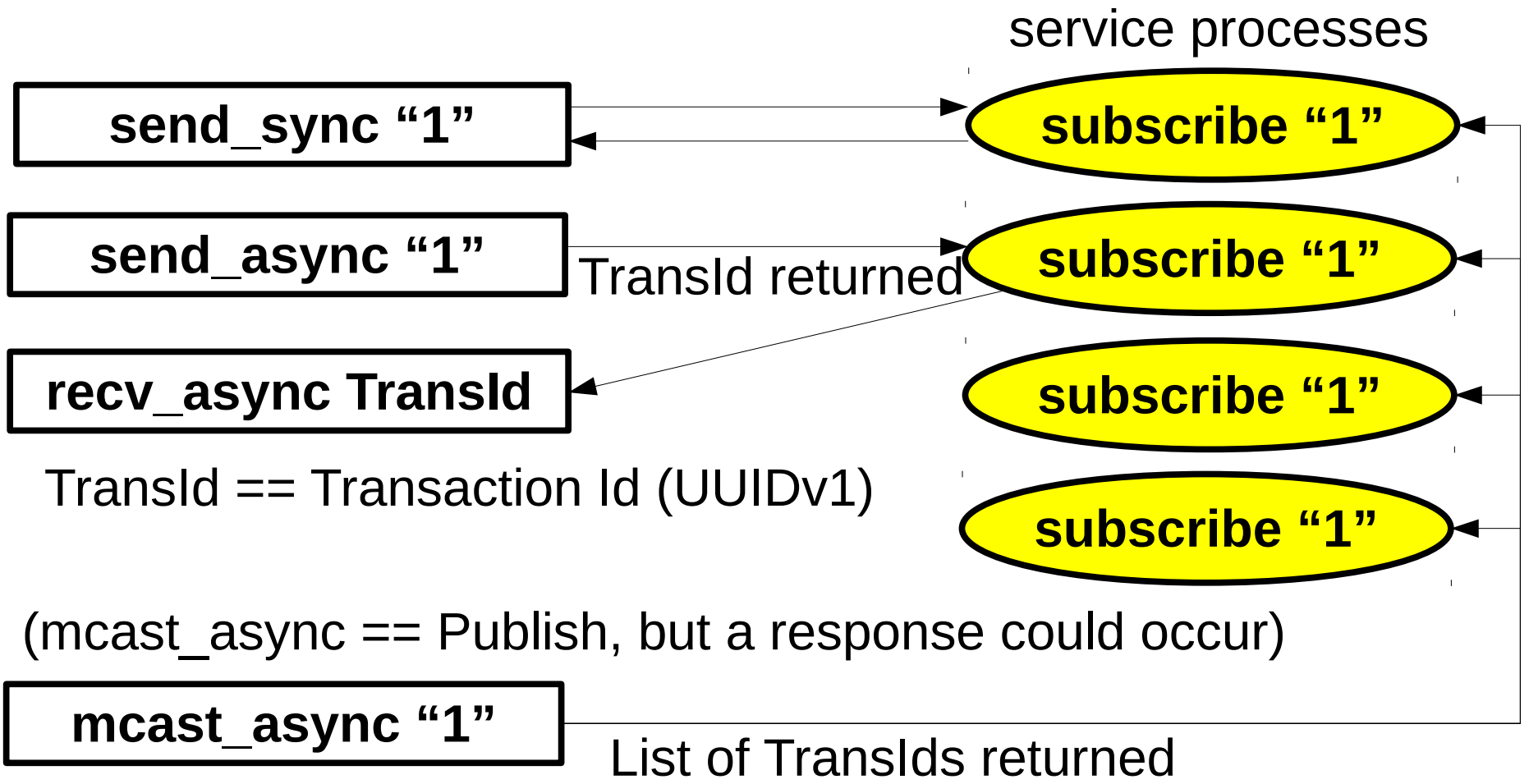
[1] http://cloudi.org/api.html#2_services_add

How do you call a CloudI Service?

- cloudi module (subset of cloudi_service module)
  → sending from any Erlang pid
- cloudi_service module
  → sending from within a CloudI Service
- Use the cloudi_service behavior when you need to receive service requests (an Erlang service is also called an “internal” service) [1]
- Each Service request is sent using: ServiceName, RequestInfo, Request, Timeout

[1] http://cloudi.org/api.html#1_Intro

Calling a CloudI Service



Service Name lookup is blocking,
so a successful return means a destination does exist

Calling a CloudI Service (cont.)

- Services are always replicated to provide fault-tolerance, no migration of state is required
- For handling N entities with services, it is best to use M service processes where M < N
(we want control of the system's scalability)
- A service request reply of “<<>>” (an empty binary, i.e., nothing) within the service is the same as the service request sender getting “{error, timeout}”
- Inversion of Control (IoC) that is more dynamic than OTP behaviors

Why is CloudI beneficial in Erlang source code?

- Dynamic fault-tolerance for many Erlang processes with one Service Name instead of being limited by Erlang's one-to-one naming of Erlang processes
- Handles memory consumption issues that are typical with long-lived Erlang processes
- Features to enforce fault-tolerance constraints and improve scalability of the service source code to simplify Erlang development
- Transaction Id is unique across all nodes

Using other nodes?

- CPG handles all the local and remote service name lookups without contacting other nodes
- hidden node connections to avoid a fully connected distributed Erlang network [1]
- automatic discovery of Erlang nodes with LAN multicast or with EC2 AWS API usage
- A service's destination refresh method [2] determines what destinations will be used for sending service requests (its view of the network)

[1] http://cloudi.org/api.html#2_nodes_set
[2] http://cloudi.org/api.html#1_Intro_dest

Result of using CloudI

- Encapsulate source code with stricter fault-tolerance constraints (doesn't persist errors)
- Easier to reuse source code (configuration driven (fail-fast)):
  → cloudi_service_queue - persistent requests
  → cloudi_service_quorum - consistency
  → cloudi_service_filesystem - file cache
  → cloudi_service_http_cowboy (and elli)
  → cloudi_service_db_pgsql (and other dbs)
- Simpler scalability

Part 2

CloudI is
Dynamic Fault-Tolerance
for non-Erlang source code

Erlang Integration Comparison

- port drivers and NIFs
  → most efficient
  → sabotages the Erlang VM's fault-tolerance (no source code is perfect)
- cnode - only a single Erlang VM connection
  → creates a bottleneck
- port - only a single pair of UNIX pipes
  → less atomic send throughput than sockets
- external CloudI service
  → a socket per configured thread

Why do we care about non-Erlang fault-tolerance?

- Why not make a bash script that restarts an OS process based on MaxR and MaxT?
  - → downtime during a restart is significant
  - → worse than 99.999% reliability (5.256 minutes per year)
  - → we want 99.9999999% reliability
- To extend the benefits of Erlang into non-Erlang source code
- To scale unscalable source code (Erlang source code can handle the scaling)

Why make an external CloudI service?

- Not everyone wants to program in Erlang
  - → Make it a CloudI service to isolate their source code with fault-tolerance constraints
  - → Scale the system from the Erlang-side
  - → Flexibility for system growth, moving to other languages or dependencies
- Usually development is feature-driven (often without clear requirements), scalability is an after-thought, fault-tolerance is impossible
- CloudI's external service integration provides practical benefits with minimal effort

Where can I find more information?

- Website
  http://cloudi.org
- Main repository
  https://github.com/CloudI/CloudI
  → examples/ - Ways of using CloudI
  → src/tests/ - Integration/Usage test examples
- Erlang-only CloudI usage with rebar
  https://github.com/CloudI/cloudi_core
- Larger integration example
  https://github.com/okeuday/sillymud