Programming Languages for Distributed Systems

From Languages to Services

Prof. P. Felber
Pascal.Felber@unine.ch
http://iiun.unine.ch/

Institut d'informatique
Université de Neuchâtel

Our Group

- "Complex Systems" research group
  - ~10 people (~7 PhD students, ~3 senior)
- Research topics
  - Dependability (fault tolerance…)
  - Distributed systems (Cloud, IoT…)
  - Concurrency (multi-core, transactional memory…)
  - Big Data (processing, storage, NVM…)
  - Energy (power-efficient computing…)
  - Security (TEEs, storage, blockchain…)
- Funding: mainly EU and Swiss projects

Our Experience with PL4DS

- SPLAY: *"distributed applications made simple"* [NSDI'09]
  - Goal: quick prototyping → real deployment/evaluation
  - Easy to read and develop (≈pseudo-code), lightweight
  - Used extensively in teaching since 2011
- Principle
  - Sandboxed environment deployed on remote nodes
  - Based on Lua (event-driven + coroutines)
  - Collection of libraries (networking, logging, serialization, storage, crypto…)
  - Reproducible experiments through controller (trace replay, churn generator with DSL)
  - Easily extensible, open-source [http://splay-project.org]

From Pseudo-code to Real Code…

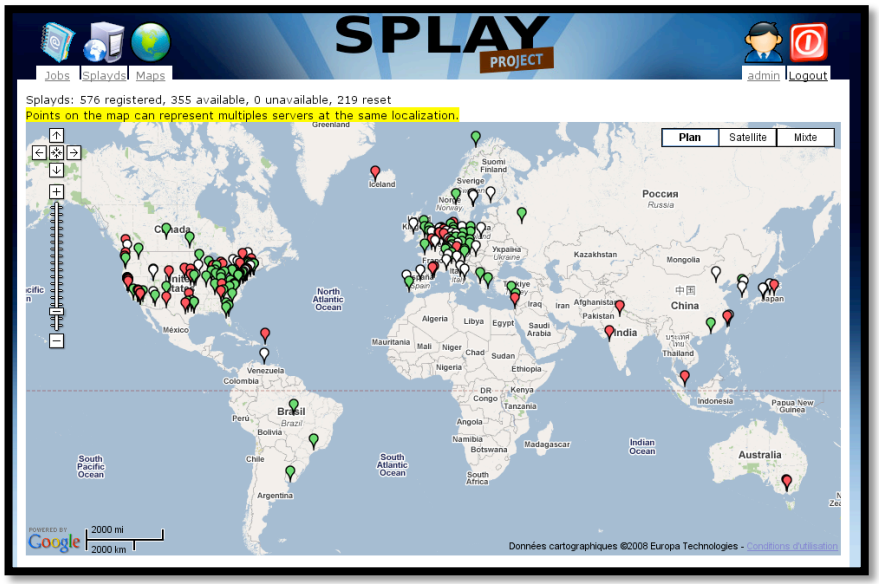
```
// join a Chord ring containing node n'.
n.join(n')
    predecessor = nil;
    successor = n'.find_successor(n);

// called periodically. verifies n's immediate
// successor, and tells the successor about n.
n.stabilize()
    x = successor.predecessor;
    if (x ∈ (n, successor))
        successor = x;
    successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
    if (predecessor is nil or n' ∈ (predecessor, n))
        predecessor = n';

// called periodically. refreshes finger table entries.
// next stores the index of the next finger to fix.
n.fix_fingers()
    next = next + 1;
    if (next > m)
        next = 1;
    finger[next] = find_successor(n + 2^(next−1));

// called periodically. checks whether predecessor has failed.
n.check_predecessor()
    if (predecessor has failed)
        predecessor = nil;
```

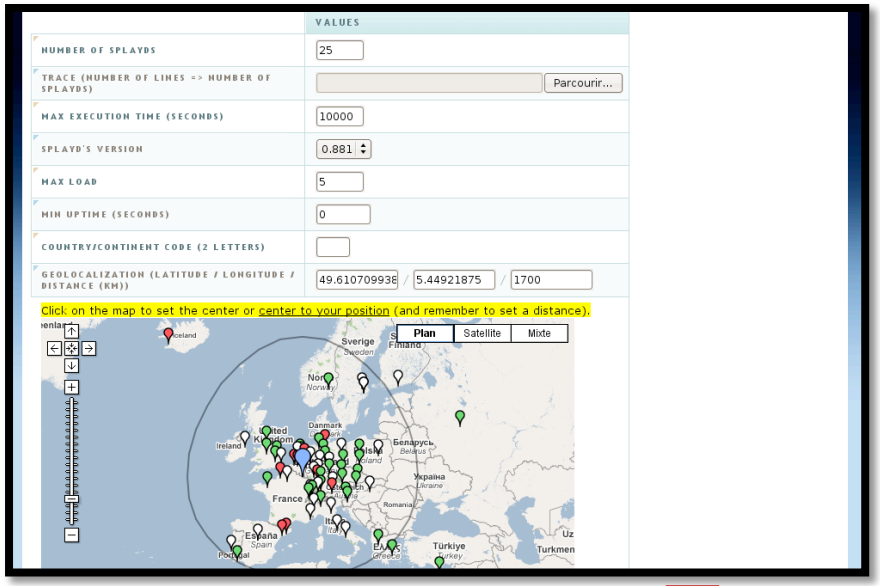
```
// ask node n to find the successor of id
n.find_successor(id)
    if (id ∈ (n, successor])
        return successor;
    else
        n' = closest_preceding_node(id);
        return n'.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
    for i = m downto 1
        if (finger[i] ∈ (n, id))
            return finger[i];
    return n;
```

```
1  function join(n0) -- n0: some node in the ring
2    predecessor = nil
3    finger[1] = rpc(n0, {'find_successor', {n.id}})
4  end
5  function stabilize() -- periodically verify n's successor
6    local x = rpc(finger[1], {'predecessor'})
7    if between(x.id, n.id, finger[1].id) then
8      finger[1] = x -- new successor
9    end
10   rpc(finger[1], {'notify', {n}})
11 end
12 function notify(n0) -- n0 thinks it might be our predecessor
13   if not predecessor or between(n0.id, predecessor.id, n.id) then
14     predecessor = n0 -- new predecessor
15   end
16 end
17 function fix_fingers() -- refresh fingers
18   next = (next % m) + 1 -- 1 ≤ next ≤ m
19   finger[next] = find_successor((n.id + 2^(next - 1)) % 2^m)
20 end
21 function check_predecessor() -- checks if predecessor has failed
22   if predecessor and not rpc(predecessor) then
23     predecessor = nil
24   end
25 end
```

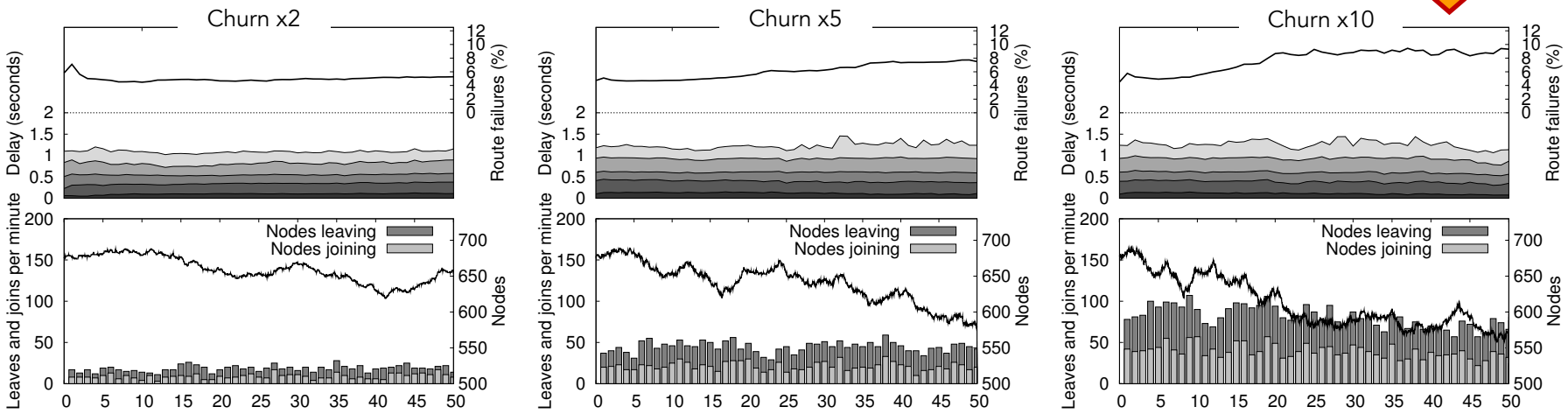
```
25 function find_successor(id) -- ask node to find id's successor
26   if between(id, n.id, (finger[1].id + 1) % 2^m) then
27     return finger[1]
28   end
29   local n0 = closest_preceding_node(id)
30   return rpc(n0, {'find_successor', {id}})
31 end
32 function closest_preceding_node(id) -- finger preceding id
33   for i = m, 1, -1 do
34     if finger[i] and between(finger[i].id, n.id, id) then
35       return finger[i]
36     end
37   end
38   return n
39 end
```

...to Deployment and Evaluation





Pastry on PlanetLab with Overnet traces



Outcome and Lessons Learned

- Several extensions
  - SplayNet: User-Space Topology Emulation [Middleware'13]
  - Teaching material, DSLs
- Pros
  - Compact, easy development, fast deployment, reproducible experiments, improved "time to paper"
  - Good for research and testing algorithms "in the wild"
- Cons
  - Corner cases hard to handle (simplicity vs. flexibility)
  - Not good enough for production systems and complete applications (but great for prototyping)
  - Limited security!

Do we Need New Languages?

- Many languages are domain-specific
  - Stream processing, dataflow, protocols, HPC, etc.
- Distributed systems are complex
  - Composition of several remote components

Single-language
vs.
composition of services
(each one developed with the most appropriate language)

Do we Need New Languages?

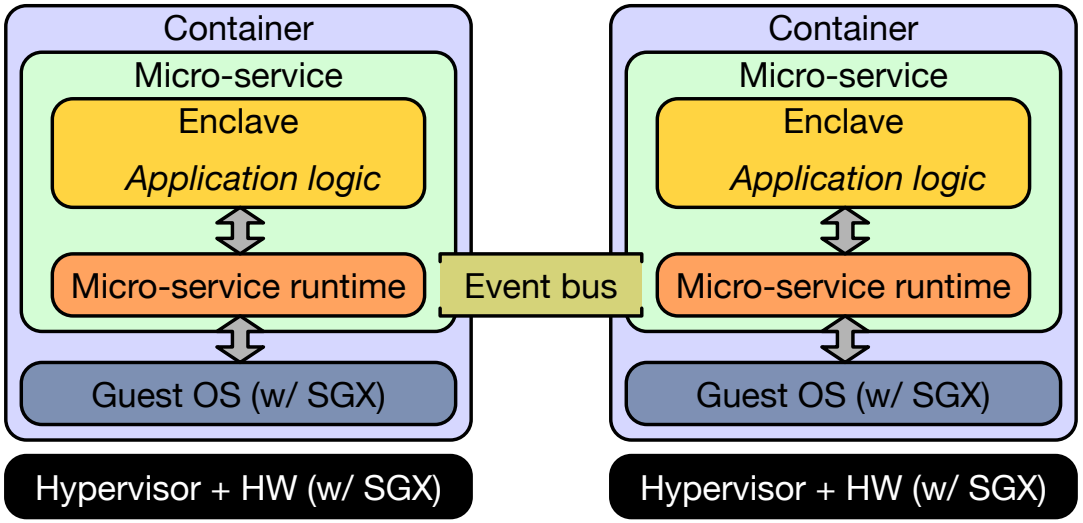
1. Microservices are great for distributed systems
   - Partition the application according to its components
   - Partition the developer teams according to their skills
   - Can be easily *containerized*
2. We need to support multiple languages with a simple way to interface the microservices
   - REST APIs are great for that: standard, interoperable
3. We need support for *"composition"* (orchestration, coordination, scheduling…)
   - Many tools available (e.g., Kubernetes, Swarm)

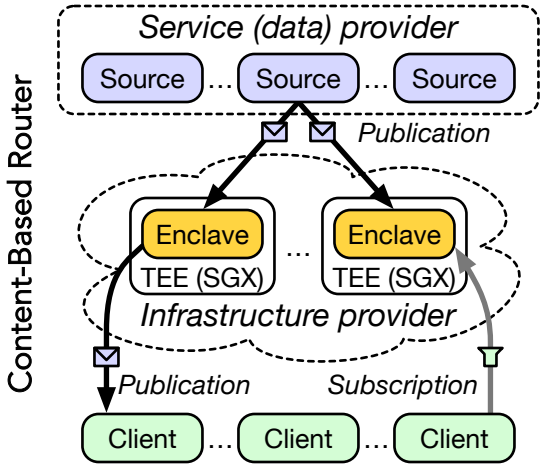
The Quest for Security

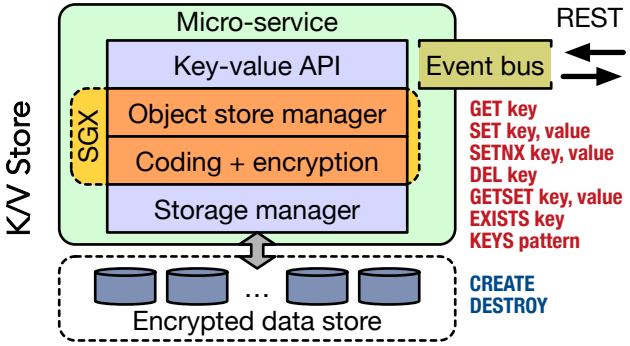
- Production systems must be protected
  - Mission-critical, vulnerable to hackers
  - Manage sensitive data
- Distributed systems are exposed
- Remote code must be secure
  - Protect the environment from the application
  - Protect the application from the environment
- PL4DS must support development of dependable (≈trustworthy) software
- Execution environment must be secure (TEE)

Our Experience with Secure µS

- SECURECLOUD: "Secure Distributed Big Data Application with Micro-Services" [H2020 EU-Brazil]
  - Micro-services within containers in TEEs in the cloud







Outcome and Lessons Learned

- Developed many μ-services (building blocks)
  - Communication, storage, processing (map-reduce)
  - Services developed in several languages (C/C++, Go, Rust, Java, Python, Lua…)
  - Run securely in containers in public clouds within TEEs
- Pros
  - Best language for each problem
  - Communication/distribution using standard tools
  - Easy composition
- Cons
  - No single language/development environment

Thanks!

DISCUSSION?