Cluster Computing at Mylife.com

- Gerd Stolpmann

  O'Caml consultant since 2005

Informatikbüro Gerd Stolpmann
http://www.gerd-stolpmann.de
http://www.camlcity.org

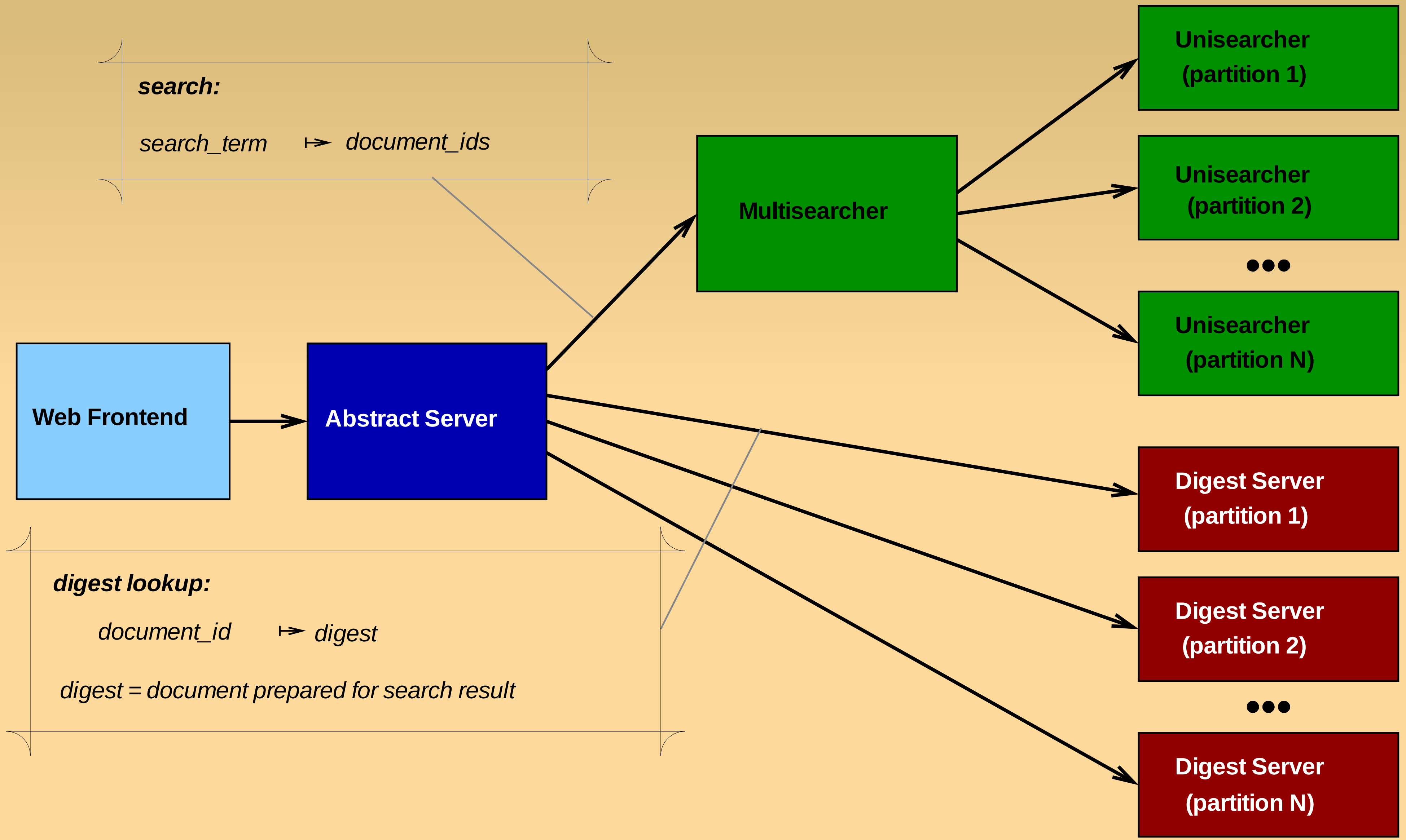
Overview

- What is Mylife.com?
- Block Diagram
- What is a cluster?
- Technologies
- Standard Components
- Error handling for asynchronous RPC
- Example: Multisearcher

Mylife.com

- People search
- Mylife.com = Reunion.com + Wink Technologies (since February 2009)
- People profiles from the web, aggregated with licensed people data
- Web sites: mylife.com, wink.com

Block Diagram Query Path



What is a cluster? (1)

- Group of machines executing together jobs, or providing together services
- Base setup of the machines is identical
- More ”power” than a single machine
- Higher availability than a single machine (in theory)
- Many components running on a cluster
- Components often deployed in a highly symmetrical way (”grid”)
- Data organization needs to be cluster-aware

What is a cluster? (2)

- Client/server architectures
  - Mylife: Remote Procedure Call
- Problems:
  - Server: How do I make myself known to others?
  - Client: How do I find the right server?
  - Client: How do I detect that the server is down?
  - Client: How do I react on a failed server?
  - Server: Parallelization
  - Client + server: Service concurrency
  - Dumb client vs. Intelligent client

What is a cluster? (3)

- Further problems:
  - Architecture: Avoid overload (”all on one”)
  - Network topology
  - How is data safely stored?

Technologies (1)

- Programming Languages
  - Ocaml: Most of our own backend programming
  - Java: Web Frontend, Lucene, Hadoop, HDFS
  - PHP: Web Frontend
- Remote Procedure Call
  - Sun RPC (only for Ocaml-Ocaml communication)
  - ZeroC ICE + Hydro (only for Ocaml-other language communication)
  - Some REST for customer APIs

Technologies (2)

- Asynchronous RPC
  - Supported by Ocamlnet implementation of SunRPC, and by Hydro
  - Client-side: useful for querying several severs at the same time
  - Server-side: useful for resource-saving implementations
- Multiprocessing
  - Generally favored over multi-threading
  - Needed for exploiting more than one core (locally, across the net)
  - Get more stable code more quickly
  - Ocamlnet-Netplex

Standard Components

- Directory and Configuration Service
  - Find service in a network
  - Confd: our own solution
  - ZeroC ICE registry
- Port Liveliness Checker
  - Is a service port alive?
  - Portchecker: for SunRPC
  - Hydromon: for Hydro
- Performance Counters
  - Perfmon
- Standard components must be rock-stable!

Error Handling

Error cases:

- RPC server impl ends with an exception
  - Solution: Log the exception, respond with an error code
- RPC call takes too long
  - Solution: Set timeout on client side
  - Different kinds of timeouts possible (next slide)
- Node is unavailable
  - Behavior 1: Router responds with ”Host unreachable” error
  - Behavior 2: No reaction at all!
  - Part of the solution: Set timeout on client side
  - Problem: Timeout cascades; distinguish from ”too long” case

Timeouts

- Socket I/O: sequence of primitive operations (*connect/send/recv/shutdown*)
- Simple timeout model:
  set timeout per I/O primitive

  However: SLAs define maximum time for user operations like *search*
- Correct timeout model:
  set timeout per user operation
- We use something in-between:
  set timeout per RPC call or complex operation

Asynchronous RPC (1)

- Defined on top of Ocamlnet's *equeue* library

```
val search :
      client → 'a → ((unit → 'b) → unit) → unit
```

- Example call:

```
search
  client
  arg
  (fun get_reply →
    try
      let r = get_reply() in
      …
    with error → …
  )
```

- Also encapsulation of such calls as *engines* possible (see `Uq_engines`)

Asynchronous RPC (2)

- Pure timers are also possible
  `Unixqueue.once tmo (fun () → ...)`
- Timeout handling:
  - Set timer
  - Start RPC call
  - When timer expires before call returns: call is canceled
  - When RPC call returns before timer expires: timer is canceled
- Cancellation of operations is essential!

Portchecker

```
val port_is_alive : Unix.sockaddr → bool
```

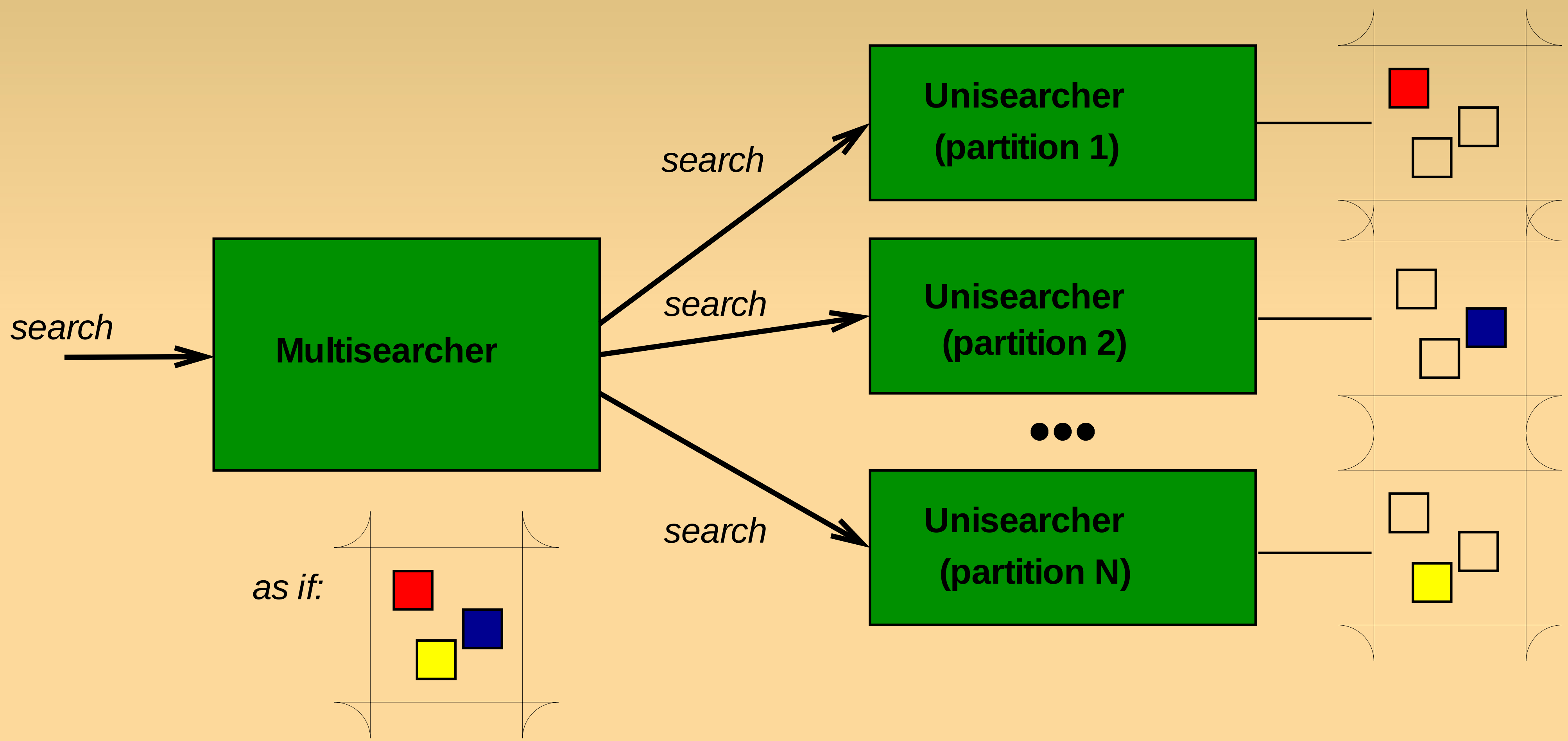
- Installed on every machine as local service
- Communication by shared memory
- Zero per-port configuration
- Starts pinging when `port_is_alive` is called
- 3 failures in sequence mean ”port is dead”
- Ping: RPC procedure 0 is called

Example: Multisearcher (1)

- Problem: Search corpus is too large for single machine
- Solution: Split it into N partitions, and put each partition on a separate machine
- Distributed search: Each user request is sent to all machines simultaneously, and results are merged
- Terminology:

  Unisearcher: the search engine for a single partition
  Multisearcher: distribution of searches

Example: Multisearcher (2)



Example: Multisearcher (3)

- For this example, assume a simple redudancy solution: each partition is installed twice, and each machine holds two distinct partitions
- Node liveliness check before each search: dead nodes are thrown out
→ Portchecker
- Timeout for the whole multisearch:
If only some nodes responded in time, take only the available results

Example: Multisearcher (4)

Implementation of multisearcher server:

```
let multisearch arg emit =
  let unisockaddresses =
    <pick sockaddress of one live unisearcher per partition> in
  let uniclients = List.map open_connection unisockaddresses in

  (* Set timer: )
  Unixqueue.once 2.0
    (fun () → List.iter close_connection uniclients);

  (* this function is called when uni results r available: *)
  let have_unisearcher_results r =
    List.iter close_connection uniclients;
    emit r
```

Continued on next slide

Example: Multisearcher (5)

```
(* Simultaneous searches on unisearchers: )
let results = ref <empty> in
let n = ref 0 in
List.iter
  (fun uniclient →
    Unisearcher.search
      uniclient
      arg
      (fun get_reply →
        ( try
            let r = get_reply() in
            results := <merge> !results r
          with error → … (* e.g. Timeout, client down *)
        );
        decr n;
        if !n = 0 then have_unisearcher_results !result
      );
    incr n
  )
  uniclients
```

The end