Lecture 6: Failure Detection and Membership, Grids
A Challenge

• You’ve been put in charge of a datacenter, and your manager has told you, “Oh no! We don’t have any failures in our datacenter!”

• Do you believe him/her?

• What would be your first responsibility?
• Build a failure detector
• What are some things that could go wrong if you didn’t do this?
Failures are the Norm

… not the exception, in datacenters.

Say, the rate of failure of one machine (OS/disk/motherboard/network, etc.) is once every 10 years (120 months) on average.

When you have 120 servers in the DC, the mean time to failure (MTTF) of the next machine is 1 month.

When you have 12,000 servers in the DC, the MTTF is about once every 7.2 hours!

Soft crashes and failures are even more frequent!
To build a failure detector

• You have a few options

1. Hire 1000 people, each to monitor one machine in the datacenter and report to you when it fails.
2. Write a failure detector program (distributed) that automatically detects failures and reports to your workstation.

Which is more preferable, and why?
Target Settings

• Process ‘group’ -based systems
  – Clouds/Datacenters
  – Replicated servers
  – Distributed databases

• Fail-stop (crash) process failures
Group Membership Service

Application Queries
  e.g., gossip, overlays, DHT’s, etc.

Membership List

Application Process $\pi$

Membership Protocol

Unreliable Communication
Two sub-protocols

Application Process $pi$

Group

Membership List

Focus of this series of lecture

Dissemination

Failure Detector

Unreliable Communication

• Complete list all the time (Strongly consistent)
  • Virtual synchrony

• Almost-Complete list (Weakly consistent)
  • Gossip-style, SWIM, ...

• Or Partial-random list (other systems)
  • SCAMP, T-MAN, Cyclon, …
Large Group: Scalability A Goal

this is us (pi)

Process Group “Members”

1000’s of processes

Unreliable Communication Network
Group Membership Protocol

I pj crashed

II Failure Detector
  Some process finds out quickly

III Dissemination

Unreliable Communication Network

Fail-stop Failures only
Next

• How do you design a group membership protocol?
I. *pj* crashes

- Nothing we can do about it!
- A frequent occurrence
- Common case rather than exception
- Frequency goes up linearly with size of datacenter
II. Distributed Failure Detectors: Desirable Properties

- **Completeness** = each failure is detected
- **Accuracy** = there is no mistaken detection
- **Speed**
  - Time to first detection of a failure
- **Scale**
  - Equal Load on each member
  - Network Message Load
Distributed Failure Detectors: Properties

- Completeness
- Accuracy
- Speed
  - Time to first detection of a failure
- Scale
  - Equal Load on each member
  - Network Message Load

Impossible together in lossy networks [Chandra and Toueg]

If possible, then can solve consensus! (but consensus is known to be unsolvable in asynchronous systems)
What Real Failure Detectors Prefer

- Completeness
- Accuracy
- Speed
  - Time to first detection of a failure
- Scale
  - Equal Load on each member
  - Network Message Load

Guaranteed

Partial/Probabilistic guarantee
What Real Failure Detectors Prefer

- Completeness
- Accuracy
- Speed
  - Time to first detection of a failure
- Scale
  - Equal Load on each member
  - Network Message Load

Guaranteed

Partial/Probabilistic guarantee

Time until *some non-faulty* process detects the failure
What Real Failure Detectors Prefer

• Completeness
• Accuracy
• Speed
  – Time to first detection of a failure
• Scale
  – Equal Load on each member
  – Network Message Load

Guaranteed

Partial/Probabilistic guarantee

Time until some non-faulty process detects the failure

No bottlenecks/single failure point
Failure Detector Properties

- Completeness
- Accuracy
- Speed
  - Time to first detection of a failure
- Scale
  - Equal Load on each member
  - Network Message Load

In spite of arbitrary simultaneous process failures
Centralized Heartbeating

- Heartbeats sent periodically
- If heartbeat not received from $pi$ within timeout, mark $pi$ as failed
Ring Heartbeating

$pi$, Heartbeat Seq. $l++$

$pi$, $pj$

Unpredictable on simultaneous multiple failures
All-to-All Heartbeating

$p_i$, Heartbeat Seq. $l++$

Equal load per member

Single hb loss $\rightarrow$ false detection
Next

• How do we increase the robustness of all-to-all heartbeating?
Gossip-style Heartbeating

Array of Heartbeat Seq. $l$ for member subset

$pi$ Good accuracy properties
Gossip-Style Failure Detection

Protocol:
- Nodes periodically gossip their membership list: pick random nodes, send it list
- On receipt, it is \textit{merged} with local membership list
- When an entry times out, member is marked as failed

<table>
<thead>
<tr>
<th></th>
<th>Address</th>
<th>Heartbeat Counter</th>
<th>Time (local)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10120</td>
<td></td>
<td>66</td>
</tr>
<tr>
<td>2</td>
<td>10103</td>
<td></td>
<td>62</td>
</tr>
<tr>
<td>3</td>
<td>10098</td>
<td></td>
<td>63</td>
</tr>
<tr>
<td>4</td>
<td>10111</td>
<td></td>
<td>65</td>
</tr>
</tbody>
</table>

Current time : 70 at node 2
(asynchronous clocks)
Gossip-Style Failure Detection

- If the heartbeat has not increased for more than $T_{\text{fail}}$ seconds, the member is considered failed.
- And after a further $T_{\text{cleanup}}$ seconds, it will delete the member from the list.
- Why an additional timeout? Why not delete right away?
Gossip-Style Failure Detection

• What if an entry pointing to a failed node is deleted right after \( T_{\text{fail}} (=24) \) seconds?

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10120</td>
<td>66</td>
</tr>
<tr>
<td>2</td>
<td>10103</td>
<td>62</td>
</tr>
<tr>
<td>3</td>
<td>10098</td>
<td>55</td>
</tr>
<tr>
<td>4</td>
<td>10111</td>
<td>65</td>
</tr>
</tbody>
</table>
```

Current time : 75 at node 2
Analysis/Discussion

• Well-known result: a gossip takes $O(\log(N))$ time to propagate.
• So: Given sufficient bandwidth, a single heartbeat takes $O(\log(N))$ time to propagate.
• So: $N$ heartbeats take:
  – $O(\log(N))$ time to propagate, if bandwidth allowed per node is allowed to be $O(N)$
  – $O(N.\log(N))$ time to propagate, if bandwidth allowed per node is only $O(1)$
  – What about $O(k)$ bandwidth?
• What happens if gossip period $T_{\text{gossip}}$ is decreased?
• What happens to $P_{\text{mistake}}$ (false positive rate) as $T_{\text{fail}}, T_{\text{cleanup}}$ is increased?
• Tradeoff: False positive rate vs. detection time vs. bandwidth
Next

• So, is this the best we can do? What is the best we can do?
Failure Detector Properties ...

• Completeness
• Accuracy
• Speed
  – Time to first detection of a failure
• Scale
  – Equal Load on each member
  – Network Message Load
...Are application-defined Requirements

• Completeness
• Accuracy
• Speed
  – Time to first detection of a failure
• Scale
  – Equal Load on each member
  – Network Message Load

Guarantee always
Probability $PM(T)$
$T$ time units
...Are application-defined Requirements

- Completeness
- Accuracy
- Speed
  - Time to first detection of a failure
- Scale
  - Equal Load on each member
  - Network Message Load

Guarantee always
Probability \( PM(T) \)
\( T \) time units

\( N*L: \) Compare this across protocols
All-to-All Heartbeating

$\pi$, Heartbeat Seq. $l++$

Every $T$ units

$L = \frac{N}{T}$
Gossip-style Heartbeating

Array of Heartbeat Seq. $l$ for member subset

Every $tg$ units = gossip period, send $O(N)$ gossip message

$T = \log N \times tg$

$L = \frac{N}{tg} = \frac{N \times \log N}{T}$
What’s the Best/Optimal we can do?

• *Worst case* load $L^*$ per member in the group (messages per second)
  – as a function of $T$, $PM(T)$, $N$
  – Independent Message Loss probability $p_{ml}$

$$L^* = \frac{\log(PM(T))}{\log(p_{ml})} \cdot \frac{1}{T}$$
Heartbeating

• Optimal L is independent of N (!)
• All-to-all and gossip-based: sub-optimal
  • L = O(N/T)
  • try to achieve simultaneous detection at all processes
  • fail to distinguish Failure Detection and Dissemination components

Can we reach this bound?

Key:
- Separate the two components
- Use a non heartbeat-based Failure Detection Component
Next

- Is there a better failure detector?
SWIM Failure Detector Protocol

Protocol period = T’ time units

K random processes

• random pj ping
• random K ping-req
Detection Time

- Prob. of being pinged in $T' = 1 - \left(1 - \frac{1}{N}\right)^{N-1} = 1 - e^{-1}$

- $E[T] = T' \cdot \frac{e}{e - 1}$

- Completeness: *Any* alive member detects failure
  - Eventually
  - By using a trick: within worst case $O(N)$ protocol periods
Accuracy, Load

- $PM(T)$ is exponential in $-K$. Also depends on $pml$ (and $pf$)
  - See paper

$$\frac{L}{L^*} < 28 \quad \frac{E[L]}{L^*} < 8$$

for up to 15 % loss rates
## SWIM Failure Detector

<table>
<thead>
<tr>
<th>Parameter</th>
<th>SWIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Detection Time</td>
<td>• Expected $\left\lfloor \frac{e}{e-1} \right\rfloor$ periods</td>
</tr>
<tr>
<td></td>
<td>• Constant (independent of group size)</td>
</tr>
<tr>
<td>Process Load</td>
<td>• Constant per period</td>
</tr>
<tr>
<td></td>
<td>• &lt; 8 L* for 15% loss</td>
</tr>
<tr>
<td>False Positive Rate</td>
<td>• Tunable (via K)</td>
</tr>
<tr>
<td></td>
<td>• Falls exponentially as load is scaled</td>
</tr>
<tr>
<td>Completeness</td>
<td>• Deterministic time-bounded</td>
</tr>
<tr>
<td></td>
<td>• Within $O(\log(N))$ periods w.h.p.</td>
</tr>
</tbody>
</table>
Time-bounded Completeness

• Key: select each membership element once as a ping target in a traversal
  – Round-robin pinging
  – Random permutation of list after each traversal

• Each failure is detected in worst case 2N-1 (local) protocol periods

• Preserves FD properties
SWIM versus Heartbeating

For Fixed:
- False Positive Rate
- Message Loss Rate

First Detection Time
- Constant

Process Load
- Constant
- O(N)

Heartbeating
- O(N)

SWIM
- O(N)

Heartbeating
Next

• How do failure detectors fit into the big picture of a group membership protocol?
• What are the missing blocks?
Group Membership Protocol

I: Process pj crashed

II: Failure Detector
   Some process finds out quickly

III: Dissemination

Unreliable Communication Network

Fail-stop Failures only
Dissemination Options

• Multicast (Hardware / IP)
  – unreliable
  – multiple simultaneous multicasts

• Point-to-point (TCP / UDP)
  – expensive

• Zero extra messages: Piggyback on Failure Detector messages
  – Infection-style Dissemination
Infection-style Dissemination

Protocol period = T time units

K random processes

Piggybacked membership information
Infection-style Dissemination

• Epidemic/Gossip style dissemination
  – After $\lambda \log(N)$ protocol periods, $N^{-(2\lambda-2)}$ processes would not have heard about an update

• Maintain a buffer of recently joined/evicted processes
  – Piggyback from this buffer
  – Prefer recent updates

• Buffer elements are garbage collected after a while
  – After $\lambda \log(N)$ protocol periods, i.e., once they’ve propagated through the system; this defines weak consistency
Suspicion Mechanism

• False detections, due to
  – Perturbed processes
  – Packet losses, e.g., from congestion

• Indirect pinging may not solve the problem

• Key: suspect a process before declaring it as failed in the group
Suspicion Mechanism

**FD**: $pi$ ping failed
- **Dissmn**: (Suspect $pj$)

**FD**: $pi$ ping success
- **Dissmn**: (Alive $pj$)

**Suspected**

- Time out
- **Dissmn**: (Failed $pj$)

**Alive**

- **Dissmn**: (Alive $pj$)

**Failed**

- **Dissmn**: (Failed $pj$)
Suspicion Mechanism

- Distinguish multiple suspicions of a process
  - Per-process *incarnation number*
  - *Inc #* for *pi* can be incremented only by *pi*
    - e.g., when it receives a (Suspect, *pi*) message
  - Somewhat similar to DSDV (routing protocol in ad-hoc nets)
- Higher inc# notifications over-ride lower inc#’s
- Within an inc#: (Suspect inc #) > (Alive, inc #)
- (Failed, inc #) overrides everything else
SWIM In Industry

• First used in Oasis/CoralCDN
• Implemented open-source by Hashicorp Inc.
  – Called “Serf”
  – Later “Consul”
• Today: Uber implemented it, uses it for failure detection in their infrastructure
  – See “ringpop” system
Wrap Up

• Failures the norm, not the exception in datacenters
• Every distributed system uses a failure detector
• Many distributed systems use a membership service

• Ring failure detection underlies
  – IBM SP2 and many other similar clusters/machines

• Gossip-style failure detection underlies
  – Amazon EC2/S3 (rumored!)
Grid Computing
“A Cloudy History of Time”

- The first datacenters!
- Timesharing Companies & Data Processing Industry
- Clouds and datacenters
- Clusters
- Grids
- PCs (not distributed!)
- Peer to peer systems
- 2012
"A Cloudy History of Time"

Data Processing Industry
- 1968: $70 M. 1978: $3.15 Billion

Timesharing Industry (1975):
• Market Share: Honeywell 34%, IBM 15%,
  Xerox 10%, CDC 10%, DEC 10%, UNIVAC 10%
• Honeywell 6000 & 635, IBM 370/168,
  Xerox 940 & Sigma 9, DEC PDP-10, UNIVAC 1108

First large datacenters: ENIAC, ORDVAC, ILLIAC
Many used vacuum tubes and mechanical relays

Berkeley NOW Project
Supercomputers
Server Farms (e.g., Oceano)

P2P Systems (90s-00s)
• Many Millions of users
• Many GB per day

• GriPhyN (1970s-80s)
• Open Science Grid and Lambda Rail (2000s)
• Globus & other standards (1990s-2000s)

Clouds
Example: Rapid Atmospheric Modeling System, ColoState U

- Hurricane Georges, 17 days in Sept 1998
  - “RAMS modeled the mesoscale convective complex that dropped so much rain, in good agreement with recorded data”
  - Used 5 km spacing instead of the usual 10 km
  - Ran on 256+ processors

- Computation-intensive computing (or HPC = high performance computing)

- Can one run such a program without access to a supercomputer?
Distributed Computing Resources

MIT

NCSA

Wisconsin
An Application Coded by a Physicist

Jobs 1 and 2 can be concurrent.
An Application Coded by a Physicist

Several GBs

May take several hours/days
4 stages of a job
  Init
  Stage in
  Execute
  Stage out
  Publish

Computation Intensive, so Massively Parallel

Output files of Job 0
Input to Job 2

Output files of Job 2
Input to Job 3

Job 2
Scheduling Problem

Wisconsin

Job 0

Job 1

Job 2

Job 3

MIT

NCSA

Allocation?

Scheduling?
2-level Scheduling Infrastructure

MIT

Wisconsin

HTCondor Protocol

Job 0

Job 1

Job 2

Job 3

NCSA

Globus Protocol

Some other intra-site protocol
Intra-site Protocol

HTCondor Protocol

Wisconsin

Job 3

Job 0

Internal Allocation & Scheduling
Monitoring
Distribution and Publishing of Files
Condor (now HTCondor)

• High-throughput computing system from U. Wisconsin Madison
• Belongs to a class of “Cycle-scavenging” systems
  – SETI@Home and Folding@Home are other systems in this category

Such systems
• Run on a lot of workstations
• When workstation is free, ask site’s central server (or Globus) for tasks
• If user hits a keystroke or mouse click, stop task
  – Either kill task or ask server to reschedule task
• Can also run on dedicated machines
Inter-site Protocol

Internal structure of different sites invisible to Globus

Globus Protocol

External Allocation & Scheduling
Stage in & Stage out of Files
Globus

- Globus Alliance involves universities, national US research labs, and some companies
- Standardized several things, especially software tools
- Separately, but related: Open Grid Forum
- Globus Alliance has developed the Globus Toolkit

http://toolkit.globus.org/toolkit/
Globus Toolkit

• Open-source
• Consists of several components
  – GridFTP: Wide-area transfer of bulk data
  – GRAM5 (Grid Resource Allocation Manager): submit, locate, cancel, and manage jobs
    • Not a scheduler
    • Globus communicates with the schedulers in intra-site protocols like HTCondor or Portable Batch System (PBS)
  – RLS (Replica Location Service): Naming service that translates from a file/dir name to a target location (or another file/dir name)
  – Libraries like XIO to provide a standard API for all Grid IO functionalities
  – Grid Security Infrastructure (GSI)
Security Issues

- Important in Grids because they are *federated*, i.e., no single entity controls the entire infrastructure

- **Single sign-on**: collective job set should require once-only user authentication
- **Mapping to local security mechanisms**: some sites use Kerberos, others using Unix
- **Delegation**: credentials to access resources inherited by subcomputations, e.g., job 0 to job 1
- **Community authorization**: e.g., third-party authentication

- These are also important in clouds, but less so because clouds are typically run under a central control
- In clouds the focus is on failures, scale, on-demand nature
Summary

- Grid computing focuses on computation-intensive computing (HPC)
- Though often federated, architecture and key concepts have a lot in common with that of clouds
- Are Grids/HPC converging towards clouds?
  - E.g., Compare OpenStack and Globus
Announcements

• MP1: Due this Sunday, demos Monday
  – VMs distributed: see Piazza
  – Demo signup sheet: soon on Piazza
  – Demo details: see Piazza
    • Make sure you print individual and total linecounts

• HW1: due next Wed 9/13! (You should have started on it already!)

• Check Piazza often! It’s where all the announcements are at!