Failure Semantics in a SOA Environment

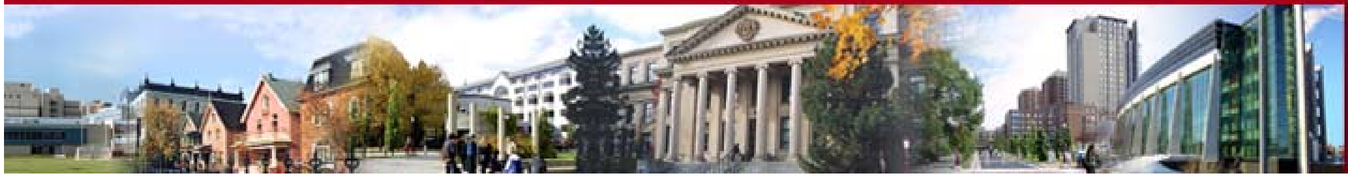
Chris Hobbs, Hanane Becha, and Daniel Amyot
damyot@site.uottawa.ca

with thanks to Abbie Barbir and Paul Knight

uOttawa

L'Université canadienne
Canada's university

Université d'Ottawa | University of Ottawa



Overview

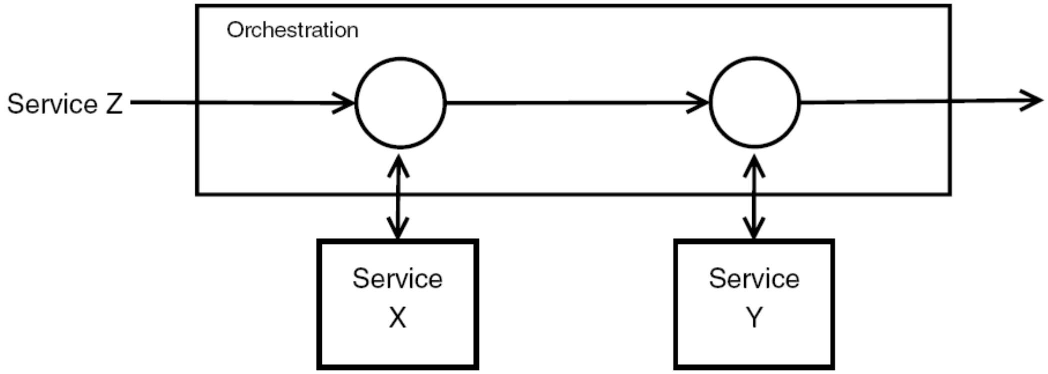
- In a Service-Oriented Architecture (SOA), services publish descriptions to permit their composition or *orchestration* into larger services.
- There are serious gaps in the semantics of SOA service descriptions, and these hinder adoption in mission-critical applications.
- We identify some of these issues and proposes a foundation for resolving one of them — *service failure*.
- The technique of *crash-only* failure is proposed as a useful first step especially for web services in a SOA.

The Crash-Only Model

- Software design approach
- Easier to restart quickly in a known state than to clean up and rebuild to recover from an error

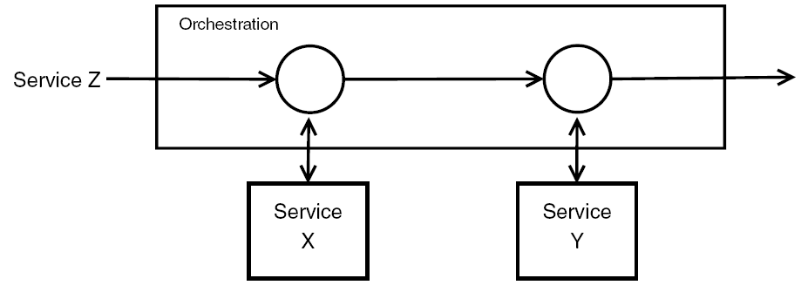
George Candea and Armando Fox are key proponents of crash-only software

Orchestration Issues



- How to characterize and guarantee service-level agreement of service Z?
  - Depends on the characteristics of services X and Y
    - X and Y might be described using different ontologies
  - Depends on the orchestration logic of service Z
  - Services X and Y are typically not owned by Z
- Difficult to test…
  - Failures of orchestrated services are often *Heisenbugs* - impervious to conventional debugging, generally non-reproducible
  - Offering service-level guarantees based on testing only is dangerous…

Orchestration Issues



- SOA specifications have provisions for specifying:
  - Interface syntax
  - Some behaviour
  - Some contracts
- But what about other relevant characteristics, more non-functional in nature?

- **Availability and Reliability**
- **Failure**
- Performance
- Management
- Security
- Privacy and Confidentiality
- Scalability
- Execution
- Internationalization
- Synchronization
- Etc.

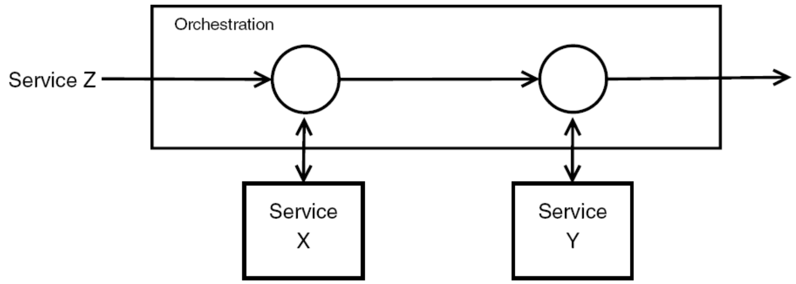
Availability and Reliability

- **Availability**: percentage of client requests to which the server responds within the time it advertised.
- **Reliability**: percentage of such server responses which return the correct answer.
- In some applications availability is more important
  - Many protocols used within the Internet are self-correcting and an occasional wrong answer is unimportant, whereas failure to give any answer can cause a major network upheaval.
- In other applications reliability is more important
  - If the service which calculates a person's annual tax return does not respond occasionally it's not a major problem - the user can try again
  - If that service respond with the wrong answer, then this could be disastrous

Availability and Reliability

- Currently, apart from raw percentage figures, these properties are hard to characterize.
  - Percentage time when the server is unavailable?
  - Percentage of requests to which it does not reply?
  - Different clients may experience these differently
    - A server which is unavailable from 00:00 to 04:00 every day can be 100% available to a client that only tries to access it in the afternoons.

Failure



- The failure models of X and Y may be very different:
  - X fails cleanly and may, because of its idempotency, immediately be called again
  - Y has more complex failure modes
  - Z will add its own failure modes to those of X and Y
  - Predicting the outcome could be very difficult
- The complexity is increased because many developers do not understand failure modeling and, even if models were to be published, their combination would be difficult due to their stochastic nature.

Failure

- One approach to describing a service's failure model:
  - Service publishes the exceptions that it can raise and associates the required consumer behaviour with each
  - "Exception D may be thrown when the database is locked by another process. Required action is to try again after a random backoff period of not less than 34ms."

- Crash-only failure model is a simple starting point for building a taxonomy of failure behaviour. This work is just beginning.

Crash-Only Software

- Historically, developers have spent a lot of effort making software resilient:
  - Put borders around it so it will not affect other things if it fails
  - Try to close it down cleanly
  - Save state
  - Reload the software component
  - Restart and replay
- Trying to keep the client from becoming aware that a failure occurred
- **Crash-only software is the opposite**
  - Client accepts that the server may crash
  - Power failure, network down, hardware, etc.
  - Client must be able to recover or restart the process by itself

Crash-Only Software Principles

- Forget recovery - more trouble than it's worth
- When the server senses a problem, it "crashes" and may perform a "micro-reboot" to return to some original state
  - e.g., a well-defined checkpoint
- The server is back working sooner than if it tried to recover via logs and journals, etc.
- Simplifies failure models, testing, and implementation

- Principles fit the Web Services paradigm nicely!
  - Loose coupling of services
  - Little state shared among services

Runtime Governance

- Intermediary between the consumer and provider of services (*management*). It has the necessary information to:
  - add *idempotency* (no need to know internal state to make decision to crash) and subscriber-dependent time-to-live information to requests to the provider.
  - monitor the provider for anomalous behaviour.
  - be the trusted source of crash commands for the provider, both as a result of delayed or insane response or as a result of a need for rejuvenation.
  - protect the provider, Z, while its crash recovery is in progress, holding off or rejecting incoming requests until recovery is complete.
  - tell the consumers when to retry.
- Can be inserted easily in SOA

Failure Description

- Published service descriptions will contain three properties:
  1. the failure and recovery type — in this case *crash-only*
  2. whether the service is idempotent or not
  3. the anticipated (modelled or measured) failure distribution

- Note that, if the service is *not* idempotent then all responsibility for determining the state of a recovered server lies with the consumer.

Failure and Availability

- Possible criticism of a crash-only architecture: potential reduction in availability!
- Crash-only paradigm effectively removes layers of sophistication built using fault-tolerant techniques
- Also trades **Mean Time to Repair** (MTTR) for **Mean Time to Failure** (MTTF).
- Regarding availability: What is 99.999% uptime?
  - 5 min 16 sec per year… but what about **distribution**?
- With crash-only, we assume it is simpler and more effective to reduce MTTR than increase MTTF.

Example [Candea et al.]

- eBid: e-Bay-like auction system
- Java application with crash-only and micro-reboot
- MTTRs between 411 and 601 msec
- For 99.999% availability, we can then allow 526 to 769 outages per year (i.e. an outage every 11 to 17 hours)
- Not difficult to meet, especially if we allow for "preventive" micro-rejuvenation during periods of low usage.

- A 769 failures/year, each with 411 msecs recovery time, better than a single failure of about 5 minutes per year?
  – Depends on the application (there are cases where both would be inappropriate), but goal likely easier to achieve.

Conclusions

- Currently, weak ways of characterizing non-functional aspects of services for enabling composition.
- For failure modes, we propose *crash-only* as a first category.
- Not adequate for all web services, but suitable for an SOA environment, and its semantic expression is relatively simple.

- Still much work remains to be done
  – Inclusion of failure information in service interfaces
  – Validation of usefulness
  – Study of other categories of failures
  – Study and integration of other qualities