Architecting For Failure
Why Cloud Architecture is Different!

Michael Stiefel
www.reliablesoftware.com
development@reliablesoftware.com

Outsource Infrastructure?

Traditional Web Application

Web Site

Virtual Machine / Directly on Hardware

100 MB Relational Database

Inbound Transactions

Output Transactions

File System

Hosting Provider Costs

Provider	**$ / Monthly Cost**
Host Gator	9.95
Go Daddy	10
ORCS Web	69
Amazon	83+ BYOS
Windows Azure	97

Note: traditional hosting, no custom colocation, virtualized data centers.

Cloud is Not Cheaper for Hosting

Perhaps, Higher Availability?

SLA is Not Radically Different

Provider	**Compute SLA (%)**
Go Daddy	99.9
ORCS Web	99.9
Host Gator	99.9
Amazon	99.95
Azure	99.95

Difference is seven minutes a day; 1.75 days a year.

Higher Rate Since You Pay for Flexibility

Hosting is Not Cloud Computing

Why Utility Computing?

Scalability: do not have to pay for peak scenarios.

Availability: can approach 100% if you want to pay.

Architecturally, they are the same problem

You must design to accommodate missing computing resources.

Designing for Failure is Cloud Computing

What’s wrong with this Code Fragment?

```
ClientProxy client = new ClientProxy();
Response response  = client.Do (request);
```

Never assume that any interface between two components always succeeds.

So You Put in a Catch Handler

```
try
{
    ClientProxy client = new ClientProxy();
    int result = client.Do (a, b, c);
}
 catch (Exception ex)
{


}
```

What if…

a timeout, how many retries?

the result is a complete failure?

the underlying hardware crashed?

you need to save the user’s data?

you are in the middle of a transaction?

What Do You Put in the Catch Handler?

```
try
{
    ClientProxy client = new ClientProxy();
    int result = client.Do (a, b, c);
}
    catch (Exception ex)
{
     ????
}
```

You can’t program yourself out of a failure.

Failure is a first-class design citizen.

Principle #1

The critical issue is how to respond to failure. The underlying infrastructure cannot guarantee availability.

Consequences of Failure

Multiple tiers and dependencies

- If your order queue fails, no orders
- If your customer service fails, no membership information

The more dependencies, the more consequences of a poorly handle failure

Dependencies include your code, third parties, the Internet/Web, anything you do not control

Unhandled failures propagate (like cracks) through your application.

Principle #2

Failures Cascade – an unhandled failure in one part of the system becomes a failure of your application.

Two Types of Failure

Transient Failure

Resource Failure

Typical Response to a Transient Failure

Retry

How Often?

How Long Before You Give Up?

Delays Cascade Just Like Failures

Delays occur while you are waiting or retrying

Delays hog resources like threads, TCP/IP ports, database connections, memory.

Since delays are usually the result of resource bottlenecks, waiting or retrying for long periods adds to the bottleneck.

Transient failures become resource failures

Transient Failures

Retry for a short time, then give up (like a circuit breaker) if unsuccessful.

Never block on I/O, timeout and assume failure.

Principle #3

There is no such thing as a transient failure. Fail fast and treat it as a resource failure.

Make Components Failure Resistant

Must Provide Failure Isolation

Make Components Failure Resistant

- Design For Beyond Largest Expected Load
  - Understand latency of adding a new resource
  - User load, virtual memory, CPU size, bandwidth, database
- Handle all Errors
- Failure affects more people than on the desktop.

Provide Failure Isolation

Catch all exceptions

Log all errors

Return Succeed / Fail to External Services

Have Failure Strategy For Dependent Services

Define your own SLA

Stress test components and system

A chain is a strong as its weakest link

Principle #4

Use a Margin of Safety when designing the resources used.

What is the cost of availability?

Any component or instance can fail – eliminate single points of failure.

Search for Dependencies

Hardware / Virtual Machines

Third Party Libraries

Internet/Web

Interfaces to your own components

TCP/IP ports

DNS Servers

Message Queues

Database Drivers

Credit Card Processors, Geocoding services, etc.

Examine Queries

Only three types of result sets:

Zero, One, Many (can become large overnight)

Search Providers limit results returned

Remember those 5 way joins your ORM uses

Objects on a DCOM or RMI call

Principle #5

Eliminate single points of failure. Accept the fact that you must build a distributed application.

You need redundancy...

but you have to manage state.

Solutions such as database mirroring may have unacceptable latencies, such as over geography.

Reduce the parts of your application that handle state to a minimum.

Loss of a stateful component usually means loss of user data.

State Handling Components

- Does the UI layer need session state?
- Business Logic, Domain Layer should be stateless
- Use queues where they make sense to hold data
- Design services for minimal dependencies
  - Pay with a customer number
  - Keep state with the message
- Don’t forget infrastructure logs, configuration files
- State is in specialized stores

Build atomic services.

Atomic means unified, not small.

Decouple the services.

Stateless components allow for scalability and redundancy.

What about the data tier?

Can you relax consistency constraints?
What is acceptable data loss?

What is the cost of an apology?

How important is the relational model?

Design for Eventual Consistency

Consider CQRS

Monitor your components.

Understand why they fail.

Reroute traffic to existing instances or another data center or geographic area?

Add more instances?

Caching or throttling can help your application run under failure.

Poorer performance may be acceptable.

Automate…Automate….Automate

Principle #6

Degrade gracefully and predictably.
Know what you can live without.

Cloud Outages Happen

Some Are Normal

Some Are Black Swans

Humans Reason About Probabilities Poorly

Principle #7

Assume the Rare Will Occur - It Will Occur

Case Study: Amazon Four Day Outage

Facts

April 21, 2011

One Day of Stabilization, Three Days of Recovery

Problems: EC2, EBS, Relational Database Service

Affected: Quora, Hootsite, Foursquare, Reddit

Unaffected: Netflix, Twillo

Why were Netflix and Twillo Unaffected?

They Designed For Failure

Netflix Explicitly Architected For Failure

Although more errors, higher latency, no increase in customer service calls or inability to find or start movies.

Key Architectural Decisions

- Stateless Services
- Data stored across isolation zones
  - Could switch to hot standby
- Had Excess Capacity (N + 1)
  - Handle large spikes or transient failures
- Used relational databases only where needed.
  - Could partition data
- Degraded Gracefully

Degraded Gracefully

Fail Fast, Aggressive Timeouts

Can degrade to lower quality service

- no personalized movie list, still can get list of available movies

Non Critical Features can be removed.

Chaos Monkey

Some Problems

Had to manually reroute traffic; use more automation in the future for failover and recovery

Round robin load balancer can overload decreased number of instances.

- May have to change auto scaling algorithm and internal load balancing.

Expand to Geographic Regions

Summary

Hosting in a cloud computing environment is valid.

Cloud Computing means designing for failure.