The Top Ten Ways to Botch an Enterprise Java™ Technology-Based Application

Cameron Purdy

President
Tangosol
www.tangosol.com

TS-5397

Overall Presentation Goal

In these slides, you will see why

That zillion dollar project failed;

You'll likely laugh—you'll want to cry!

To see these practices revealed

Speaker’s Qualifications

- Cameron Purdy is President of Tangosol, and is a contributor to Java™ technology and XML specifications
- Tangosol is the JCache (JSR 107) specification lead and a member of the Work Manager (JSR 237) expert group
- Tangosol Coherence is the leading clustered caching and data grid product for Java and J2EE™ platform environments; Coherence enables highly scalable in-memory data management and caching for clustered Java technology-based applications

Disclaimer

Common Sense Trumps Dogma

- There will be real-world situations in which the principles from this presentation will be wrong; Always use common sense
- Any similarity of material in this presentation to disasters that you have witnessed, either real or imagined, is purely coincidental

The Top Ten Ways to Botch an Enterprise Java Technology-Based Application

#10

Specifying the mechanism for data access without understanding the granularity of the data model

Popular Methods for Data Access

- Java Database Connectivity (JDBC™)—
  a CLI view of RDBMS data
- Apache iBATIS—simplifying common JDBC API usage patterns
- Enterprise JavaBeans™ (EJB™) architecture
  v1, v2—a “record oriented” approach
- Object Relational Mapping (ORM)
  - Hibernate, Castor
  - Java Data Objects (JDO) v2 Including KODO, OpenAccess)
  - EJB v3 technology (Including Hibernate, Toplink, KODO)

Data Access: JDBC API

- The “assembly language” of RDBMS—you can do anything, but you have to do everything
- Best choice when the form of the data being accessed is unknown, such as in a reporting engine in which the number and types of result columns are unknown
- Good choice for dealing with extremely large result sets and accessing rarely-used driver functionality
- Worst choice for rapid application development
- Worst choice for large engineering teams and large code bases
- Worst choice for building maintainable applications

Data Access: ORM

- The “object oriented” model for RDBMS—
you only deal with objects, but the ORM has to deal with the RDBMS
- Best choice when the form of the data being accessed is well known, and the widespread use of the data throughout the application logic far exceeds the investment in defining the object schema and its mapping to the database
- Good choice for enabling data caching
- Definitely **not** a silver bullet
    - Still requires good development processes and careful design
    - Using ORM, some common application use cases are very inefficient compared to hand-coded JDBC API

Why the Choice Is So Critical

- I have witnessed more applications fail to meet their business goals due to poor choices around data access than any other category
- Once a choice is made, it tends to be reflected in every aspect of the application, making later changes more difficult and incredibly costly
- The predictability and cost of scalability of a large scale application is tightly bound to the application’s data access model

Choosing a Data Access Model

- Understand the high-level requirements
  - The “-ilities”: Scalability, Reliability, Availability,…
- Visualize the data flows in the running system
  - For each page or service request, what actually goes through to the database, and why?
- Understand the impact of concurrent users
  - How will database contention be minimized?
  - How will cache effectiveness be maximized?
- Understand the application’s data granularity
  - Set-centric or identity-centric?

The Top Ten Ways to Botch an Enterprise Java Technology-Based Application

#9

Assuming the container will take care of transactions

The Container Can Take Care of It

- Java Platform, Enterprise Edition (Java EE) containers are designed to manage transactions
  - Database transactions (JDBC API)
  - Messaging transactions (Java Message Service)
  - Other EIS (Java Cryptography Architecture)
  - Coordination over multiple transactional systems (XA)
- The problem is in the assumption
  - Ignorance will cost you zillions
  - XA 2PC can have a staggering latency and throughput cost
  - Misapplication of transactions could result in a failure to achieve business requirements such as reliability
- Goal: Understand transactional requirements

Leveraging the Container

- Declarative transactions
  - To ensure “normal” transactional behavior, use “Required”
  - Advanced: To explicitly guarantee a separate transaction within another transaction, use “RequiresNew”
  - For RDBMS-based applications, all other options are verboten
  - Trying to avoid the cost of a transaction by using “Supports”, “NotSupported” and “Never” will almost certainly back-fire
- XA: If multiple transactional resources are required within a transaction, Java technology can do it
  - Unfortunately, in relative terms XA is not very well-known
  - Don’t assume that your container actually supports it—test!
  - Documentation, examples and FAQs are hard to find

Alternatives to Traditional 2PC

- Combine multiple resources
  - Put the JMS store on the same database server
- Compensating transactions
  - Useful for resources that are not truly transactional
- Idempotency
  - Allows blind retries of any request without knowing for certain the outcome of previous attempts to process that request
  - When it is applicable, idempotency is a silver bullet

The Top Ten Ways to Botch an Enterprise Java Technology-Based Application

#8

Using a stateless architecture

"Stateless": The Intention

- The stateless concept was originally intended as a server-side optimization for conversational (stateful) interfaces and protocols
- To accomplish statelessness, the conversational state must be included in every request, and often in every response
- Particularly useful for "connectionless" network protocols, e.g., HTTP
  - For example, "cookies" are used in the HTTP request/response headers to encapsulate conversational state

When to Use “Stateless”

- You are writing an HTTP server
- Your application has no security requirements
- The only data that your application needs for its operation will be passed to it
  - For example, the mythical “number addition service”
- Scalability is not a requirement
- Your budget is infinite

“Stateless”: The Reality

- If scalability is a requirement, then a stateless architecture will only work well for a truly stateless application
- When the term stateless is used to describe a Java EE platform-based application, it usually indicates a “stateless middle tier”
- If an application has a database or an EIS, then there is state to be managed
- If there is state to be managed, then stateless applications are simply “passing the buck” to a more expensive tier

“Stateful”: The Solution

- There are degrees of statefulness
  - Goal is to avoid increasing load on underlying services while avoiding unnecessary application complexity
- Simple
  - Caching read-only data
  - HTTP Sessions
- Intermediate
  - Caching read/write data
  - Client state managed server-side using Stateful Session EJB bean
- Advanced
  - Transactional caching
  - Stateful application-level services

The Top Ten Ways to Botch an Enterprise Java Technology-Based Application

#7

Designing the application for deployment on a single server and leave scalability and reliability up to the container

Scale-Out Limitations

- In large scale systems servicing concurrent requests, scalable performance is limited by the number of operations which must utilize any shared resource that does not exhibit linear scalability
  - Databases, EIS, mainframe services
- A stateful application without an explicit requirement for a scale-out architecture will not scale out, or will scale out very poorly
  - Scaling out quickly highlights bottlenecks in shared resources, particularly related to state management

Scale-Out Gotchas

- Common architectural patterns fail
  - The singleton pattern (isn’t!)
- Stateful applications have additional considerations in a scale-out environment
  - Clustered caching
  - Distributed state management
- Reliability of data requires concurrency management and transactions
  - Yet relying on a central store will create a bottleneck
  - Data Grids and Information Fabrics

Designing for Scale-Out

- Cookie Cutter: Every server has identical responsibilities
  - Benefit: The same application works on one server or one hundred servers
- Conceptually, there are only two things that can move in a distributed environment
  - State—application data
  - Behavior—application processing
- Locality of information is the critical enabler
  - Move as little as possible, as rarely as possible
  - Always avoid “going to the committee”

The Top Ten Ways to Botch an Enterprise Java Technology-Based Application

#6

Utilizing popular technologies such as Web Services for component integration and remoting

XML Services: A Brief Synopsis

- XML documents: Rendering a request or response as an XML document tends to be both CPU and memory intensive
- Networking: Passing textually-encoded XML documents is a poor use of bandwidth
- Connection Management: Connectionless HTTP is far more expensive than a socket
- Parsing: Validation and parsing of XML documents is extremely CPU intensive
- Summary: Big money, high latency

Appropriate Uses

- Exposing Platform-Independent Services
    - XML-based services are the one thing that Java technology and Microsoft .NET can agree on
- Loose Coupling
    - The service provider and the service consumer can be completely unknown to each other
- Public Network Capable
    - Securable at the low level (e.g., HTTPS) and the high level (document signing, encryption, etc.)
- Defensible Uses
    - Among separate applications
    - Among different organizations

Yes, SOA Is Still a Good Thing

- XML-based SOA has appropriate uses
- SOA and its underlying principles do not require SOAP, XML, HTTP, etc.
- Java technology has a rich set of built-in capabilities for supporting SOA within an application, such as
  - JMS API—Message bus-based SOA
  - RMI—Synchronous Java-centric remote invocation
- When the internal (Java technology) and external (XML) service interfaces are identical, then layer the XML-based interfaces on top of the higher-performing and more efficient Java API, and use the Java API within the app

The Top Ten Ways to Botch an Enterprise Java Technology-Based Application

#5

Rolling your own frameworks

What Business Are You In?

- Whenever you realize that you are rolling your own framework, ask yourself if you are in the framework business
- Be prepared: It is almost always provable that no existing framework is a perfect fit to your specific problems
  - Ignore the temptation to write your own
  - If others have been able to use existing frameworks, then so can you!
- Existing frameworks benefit from cumulative experience, testing and refinement

Solution: Don’t Roll Your Own

- Look at frameworks—even free ones—the same way you would look at any build versus buy decision
- Adopt a “best of breed” approach to selection
  - Technical excellence
  - Strong market presence
  - Ongoing commitment to support
- When requirements differ substantially from what is already available, consider starting from an existing framework and customizing it

The Top Ten Ways to Botch an Enterprise Java Technology-Based Application

#4

Distributing synchronous object graphs across servers

First Law of Distributed Objects

- “Don’t distribute your objects!”

  – Martin Fowler
- Co-location of state and behavior is the lynch-pin of scalable performance
  - Move as little as possible, as rarely as possible
- “Transparency is valuable, but while many things can be made transparent in distributed objects, performance isn’t usually one of them.”

  – Martin Fowler, in Errant Architectures

Synchronous Distribution

- Synchronous execution in a distributed environment can turn a collection of servers into a single thread—and an inefficient one at that!
- When distribution is necessary, minimize it
  - Use replication to “push” data, events to “push” data invalidations, and pinned services for “pull”
- Use JMS API for asynchronous distribution
  - Removes the distribution from the synchronous application flow
  - Warning! Durable JMS API has huge implications (XA, synchronous logging)

The Top Ten Ways to Botch an Enterprise Java Technology-Based Application

#3

Designing logic and data flows assuming the application is a single-user system

Single-User Systems

- No need to worry about transaction isolation or concurrency strategies
    - Transactions never deadlock
    - Optimistic transactions always commit.
    - Transactions are always Serializable…because they are serial!
- Scalability never suffers from contention
- Data sharing is never needed
- Shared resources are always completely free!

Designing for Concurrent Load

- Biggest cause of failure in large-scale application development: The fast PC!
  - Developers feel that the application is plenty fast
- Developing for concurrent load is a conscious decision
  - Developers usually test using a single user approach
  - Developers do not see the throughput impact of multiple users
  - Developers do not witness the issues related to concurrency control, such as transaction roll-backs

Designing for Concurrent Load

- Applications developed for high levels of concurrent load have different priorities
  - Code profiling and other forms of micro-optimization are much less important than system profiling
  - Maximizing the performance of each request takes a back seat to maximizing the Scaling Factor (SF)
  - Linear scalability (SF=1.0) is a lot more important—and a lot harder to achieve—than the fastest single-user response time!
  - Use of shared resources, such as databases, is considered to be extremely expensive

The Top Ten Ways to Botch an Enterprise Java Technology-Based Application

#2

Compensating for a lack of knowledge of the application domain by building in systemic flexibility

Unnecessary Flexibility

- Maintaining flexibility often means giving up scalability and/or reliability
  - Over-normalizing the data schemas
  - Using distributed transactions when local will suffice
  - Using transactionally consistent data when it's not required
- Flexibility tends to oppose efficient execution
  - Bulk pass-through operations get turned into iterators
- Flexibility is good—when it is necessary
  - You can only say “YAGNI” if you have a good understanding of the problem domain!

The Top Ten Ways to Botch an Enterprise Java Technology-Based Application

#1

Putting off system testing until the application is ready to deploy

Test Early, Test Often

- Obviously you don’t want to design in the problems in the first place
- But if you do, the earlier you realize it’s a disaster, the better
- Make the tests reflect real world use cases
  - Closely simulate the production environment
  - Load test with multiple users, testing latency and throughput
  - Simulate failure scenarios, failover and failback—under load!

Eliminating SPOFs

Eliminating SPOFs: The Java EE Platform Tier

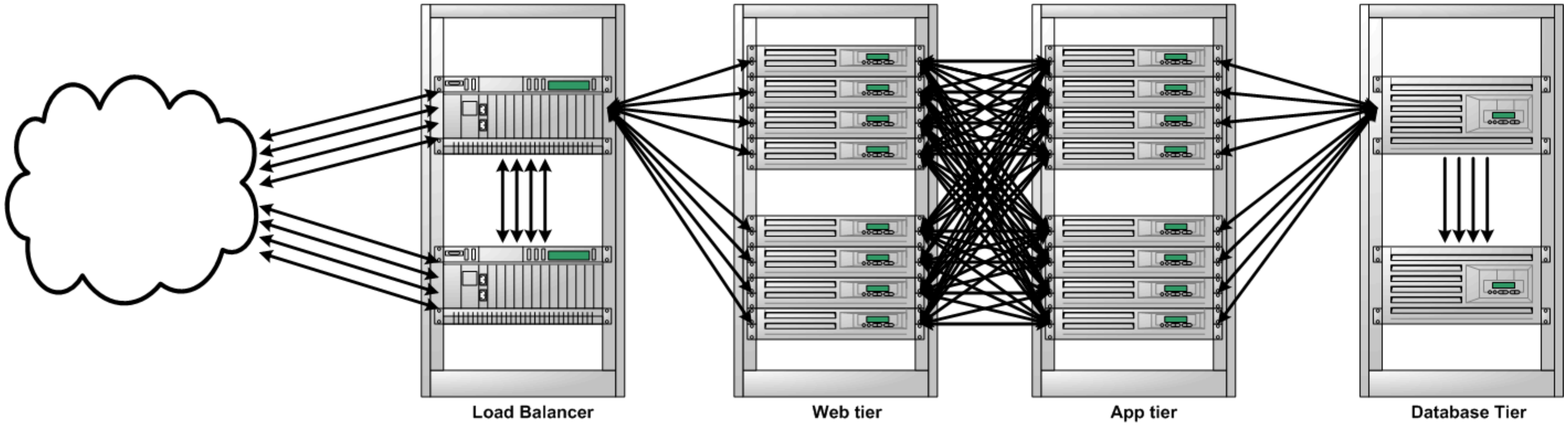
- Java technology-based application tiers ("Java tier") can be stateless
  - Stateless tiers (e.g., web servers) are HA using simple redundancy
  - Only problem is that statelessness in one tier usually just passes the buck to the next tier, which is almost always more expensive

Eliminating SPOFs: The Java EE Platform Tier

- Java tiers are almost always stateful
  - Only two things can be lost: State and inflight requests
  - To achieve HA, the Java tier must either manage its state resiliently (e.g., in a clustered coherent cache) or back it up to a central store
  - Idempotent actions can be replayed by the web tier when a server fails

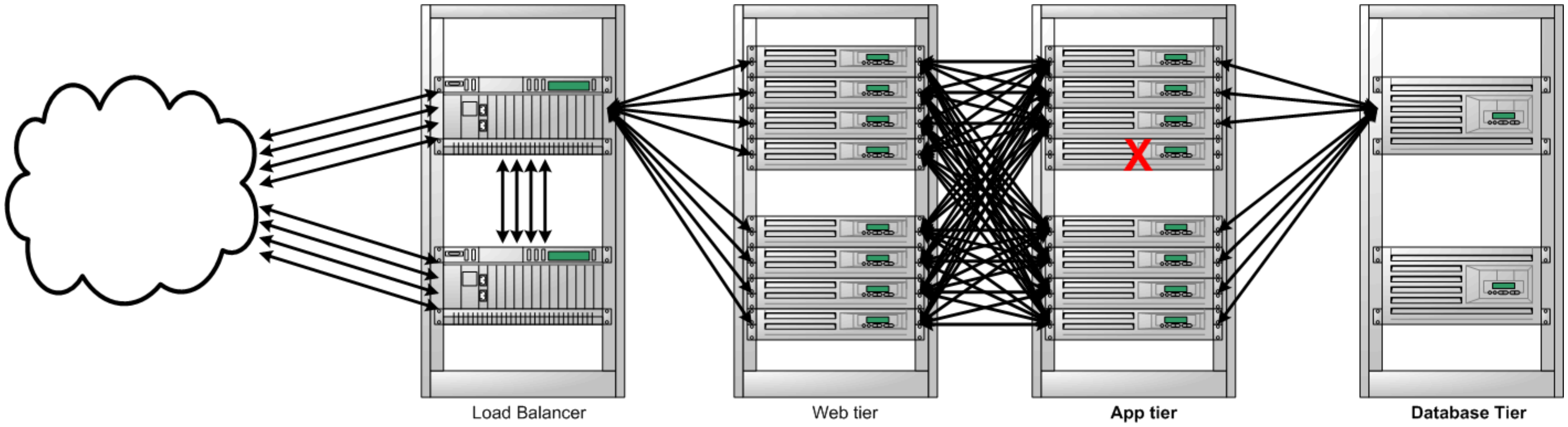
Eliminating SPOFs: HA Java EE Platform

- Normal flow through a multi-tier enterprise app…



Eliminating SPOFs: HA Java EE Platform

- …if a server dies, its current requests can be lost



Web Tier to App Tier Interconnects

- The previous pictures may look like a mess…
  - …but load balancers have to do far more work to support sticky load balancing
  - Best approach is for the load balancer to round-robin or randomize its load-balancing across all available web servers

Web Tier to App Tier Interconnects

- …but there's a good reason
  - Web servers (e.g., Apache, IIS, Java Enterprise System) can handle lots of concurrent connections, serve static content, and route requests to app servers
  - The web server plug-in for routing to the app server can do the sticky load balancing, guaranteeing that HTTP Sessions "stick"!

Eliminating SPOFs: Idempotency in Java EE Platform

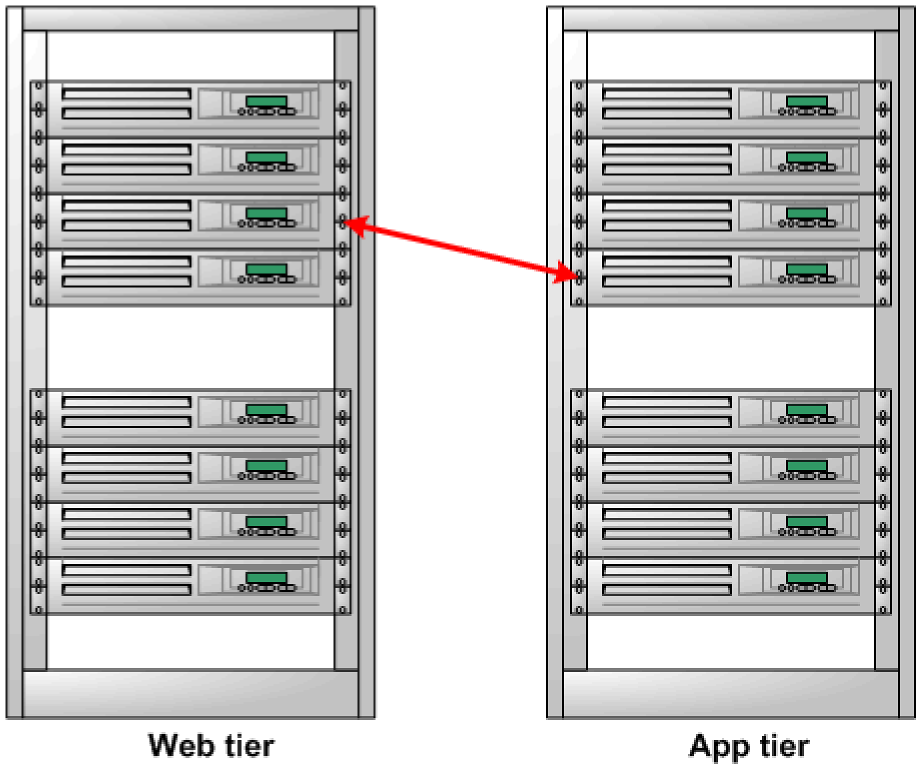
- Idempotency is potent!
  - Under normal conditions, requests are processed exactly once
  - Idempotency allows the same request to be processed more than once, with the possibility that those requests were partially processed, and without any side-effects from being run more than once

Eliminating SPOFs: Idempotency in Java EE Platform

- Idempotency is potent!
  - Allows blind retries of any request without knowing for certain the outcome of previous attempts to process that request
  - Requires great forethought: every potentially state-mutating request must have a plan for how it can be run 1 time, 1.5 times, 2 times or 200 times without corrupting the application state

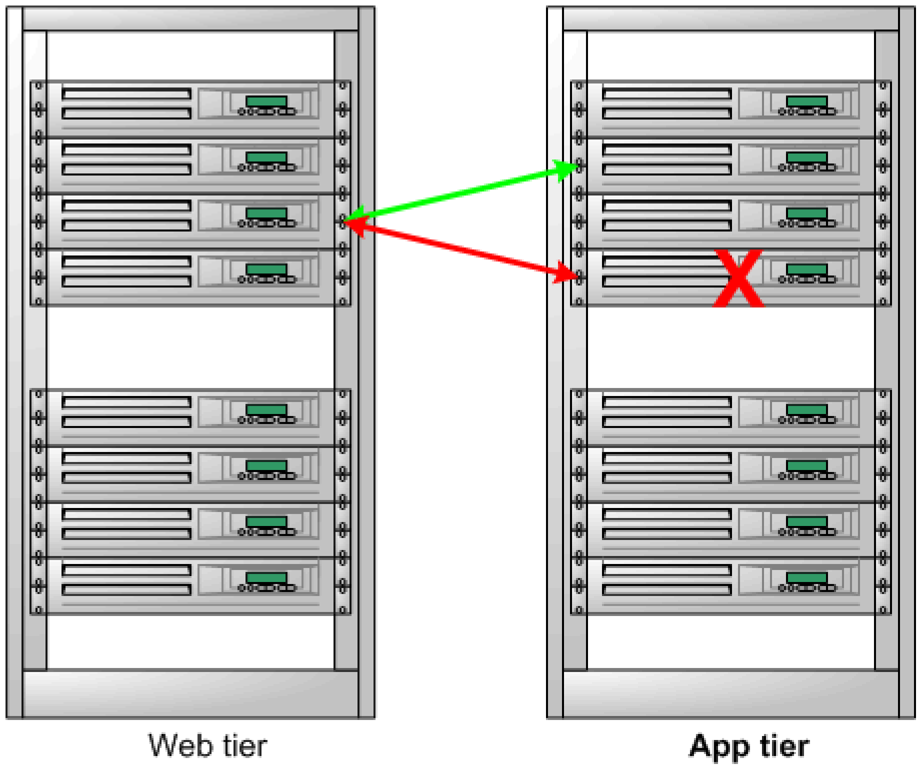
Eliminating SPOFs: HA Java EE Platform

- Normal request/response before a server dies…



Eliminating SPOFs: HA Java EE Platform

- …and with idempotency, requests can re-route!



Eliminating SPOFs: Idempotency in Java EE Platform

- Idempotency by predicate
  - All actions must have one-way non-destructive state transitions
  - Conceptually similar to optimistic concurrency with a database
  - e.g., “Perform this account transfer of $100 from account 123 to account 456, but only if account 123 contains exactly $1000 and account 456 contains exactly $500”

Eliminating SPOFs: Idempotency in Java EE Platform

- Idempotency by identity
  - Easy pattern: Uniquely identify each possible action **before** it occurs
  - It's like the command pattern, but every command instance has an UID
  - e.g., "Place this order for these goods, but only if order UID 1234567890 has not previously been submitted."
  - Bonus: Allows the user to click submit twice without their credit card getting charged twice!

Evil Rules

Designing Non-Scalable Applications

- Evil Rule 1: Create SPOBs
  - A Single Points Of Bottleneck (SPOB) is any server, service, etc. that all (or many) requests have to go through, and that has any load-associated latency
  - The simplest way to create a SPOB is to read data from a shared database as part of request processing
  - Web services, mainframes, enterprise applications such as PeopleSoft and SAP, singleton distributed services, etc. all provide great opportunities for introducing SPOBs

Designing Non-Scalable Applications

- Evil Rule 2: Introduce concurrency control bottlenecks
  - Default to pessimistic concurrency, and hold locks on the database whenever possible
  - Default to serializable transactions
  - In a multi-threaded application, make sure to synchronize on some shared object before making a database call or a web service invocation
  - Use synchronous logging

Designing Non-Scalable Applications

- Evil Rule 3: Build in extra tiers and remote invocations whenever possible
  - Never miss an opportunity to split something out as a remote web service
  - Make sure that the JavaServer Pages™ (JSP™) specification-based pages and Servlets make remote calls to the EJB technology tier
  - Treat remote objects as if they were local
  - Never do in a single SQL statement what you could spread across a whole bunch of individual statements

Designing Non-Scalable Applications

- Evil Rule 4: Push more work onto the expensive parts of the infrastructure
  - Make sure that the database is a SPOB, since it has a typical SF between 0.70 and 0.90 and an exponentially increasing cost factor for CPU scaling
  - Mainframes are big and fast, so don’t worry about calling the same service more than once with the same request parameters

Q&A

Cameron Purdy

The Top Ten Ways to Botch an Enterprise Java™ Technology-based Application

Cameron Purdy

President
Tangosol
www.tangosol.com

TS-5397