Diagnosability for Distributed Applications

Peng Dai

Why Are You Here

- Developers
  - Programming abstractions for simplified distributed diagnosis
  - Failure modes and consistency semantics
  - How does DSP facilitate with exception handling
- QA
  - Failure modes of DSP based applications
- Support
  - Diagnostics of DSP based applications
- Others
  - General interest in distributed systems

Design Objectives

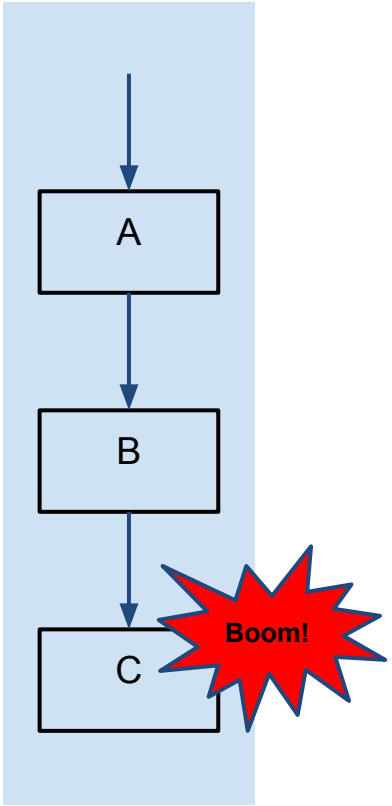
- To enable writing “distributed applications”
- Handle exceptions transparently
  - Exceptions handled in the infrastructure will remain there
- Make exceptions easier to diagnose
  - For those that must escape the infrastructure and go up the stack
  - Exceptions pop up in different locations but diagnosis doesn’t have to be
- Avoid new programming paradigms
  - New paradigms are costly to learn
  - Existing “local” paradigm preferred (synchronous, asynchronous, …)

Exception Classification

- Application
  - Tied to specific application semantics
  - Application involvement required for handling
  - Applicable to both local and distributed
  - Benefit from diagnosability improvement
- Infrastructure
  - Environment related such as system, network, and what not
  - Application agnostic
  - Unique to distributed applications
  - Benefit from transparent handling

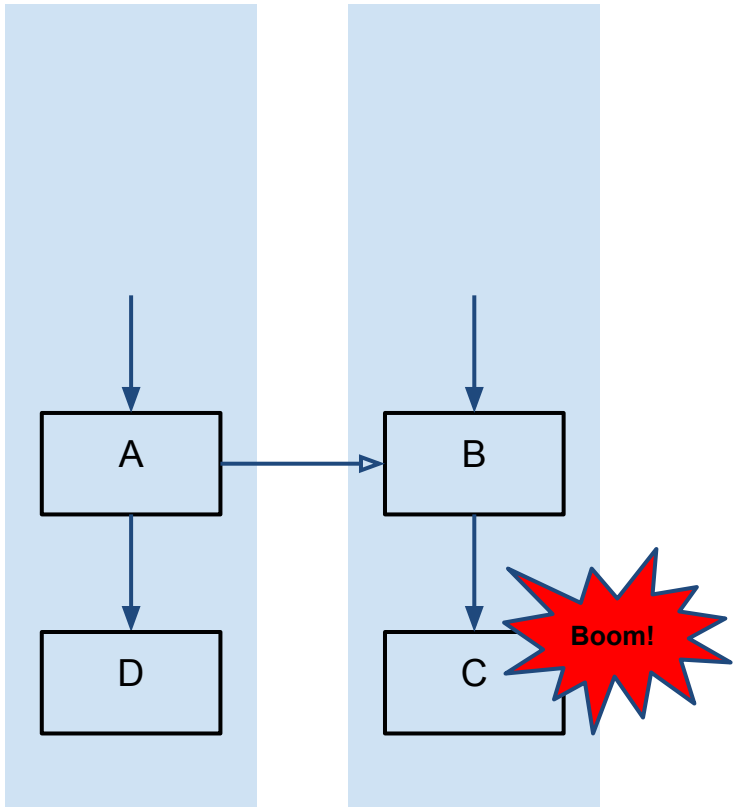
Synchronous Exception

- The symptom is the failure to execute A
- The root cause is the exception hit in C
- The diagnosis is represented by the causality chain { A -> B -> C }
- The diagnosis is fully expressed by the exception and the corresponding stack trace



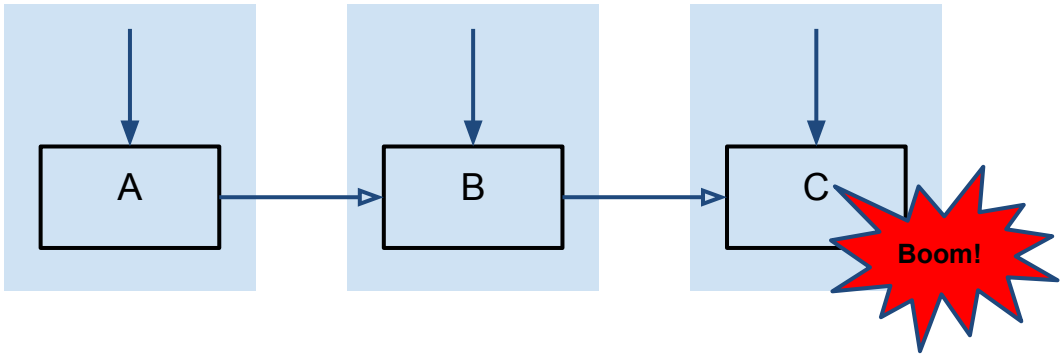
Asynchronous Exception

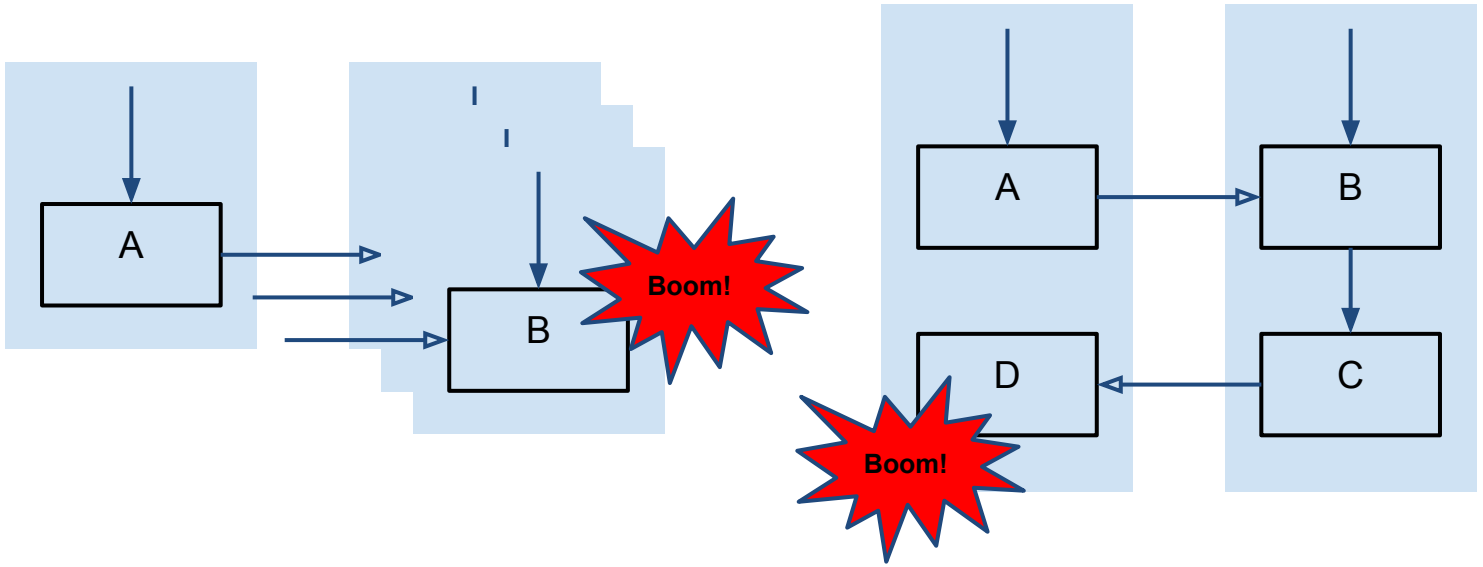
- *The symptom is the failure to execute A*
- *The root cause is the exception hit in C*
- *The diagnosis is represented by the causality chain { A -> B -> C }*
- The exception and the corresponding stack trace captures only **part** of the causality chain { B -> C }



Execution Topologies

- Serial { A -> B -> C }
- Parallel { A -> { B, C, D } }
- Cyclical { A -> B -> A }





Complications

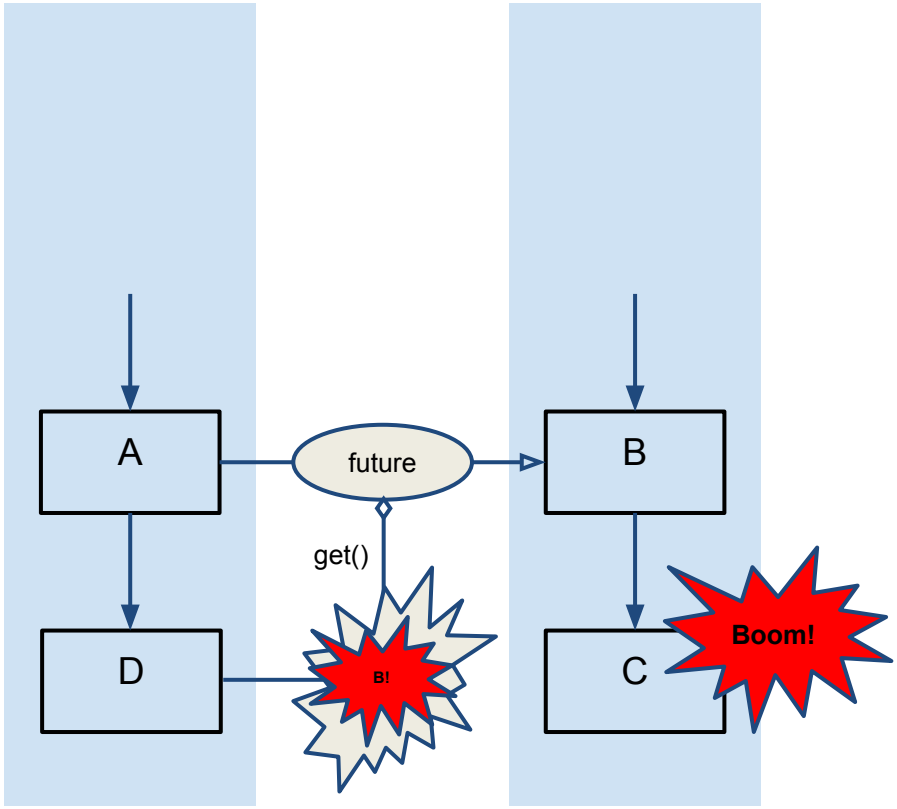
- Exception occurred in an asynchronous context is not automatically propagated across execution context boundaries
- Asynchronous execution has complex topology in general and that may further complicate diagnosis, f.g.,
  - Multiple exceptions may occur in different asynchronous contexts simultaneously, leading to “race” in diagnosis
  - ...
- Exception occurred in one asynchronous context may affect the normal execution of others
  - One implication of this is the need for task management

Language Support

- Pre Java 5 offered a thread centric approach to concurrent programming with primitives such as Thread and Runnable
  - Lack of support for asynchronous result, exception, task management
  - Explicit management of execution context required
- Java 5 offered a task centric approach to concurrent programming with primitives such as Callable, Future, and ExecutorService
  - Callable adds task result support to Runnable
  - Future refers to the asynchronous task execution state
    - Task state inquiry via isDone() and isCancelled()
    - Result or exception retrieval via get()
    - Active task management via cancel()
  - ExecutorService hides the details of execution context management

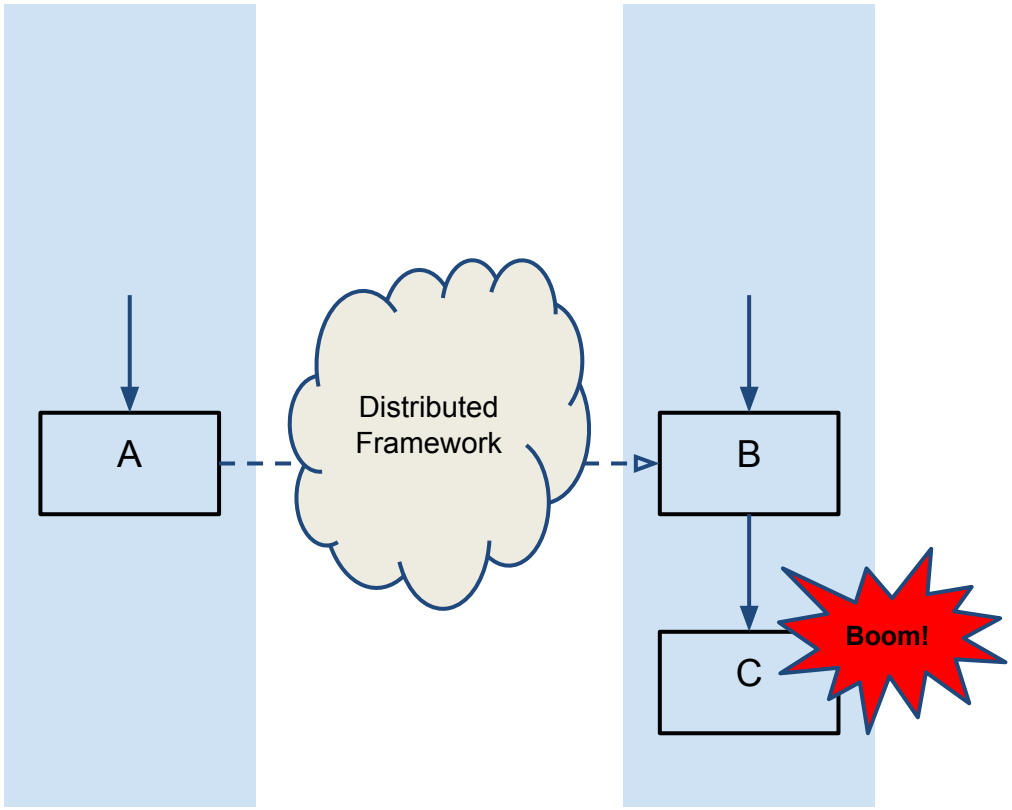
Asynchronous Exception Revisited

- A future is returned when the asynchronous task is submitted for execution
- future.get() is called later to retrieve asynchronous task result
- Asynchronous exception is wrapped in a checked exception and thrown from get()
- A diagnosis may be obtained by stitching together the inquiry site and the asynchronous exception, or { D -> B -> C }



Distributed Exception

- Synchronous to asynchronous is no less of a paradigm shift
- Asynchronous to distributed doesn't have to be
- Asynchronous model is a natural fit for distributed systems
- Onus falls on the underlying distributed framework to hide the delivery specifics



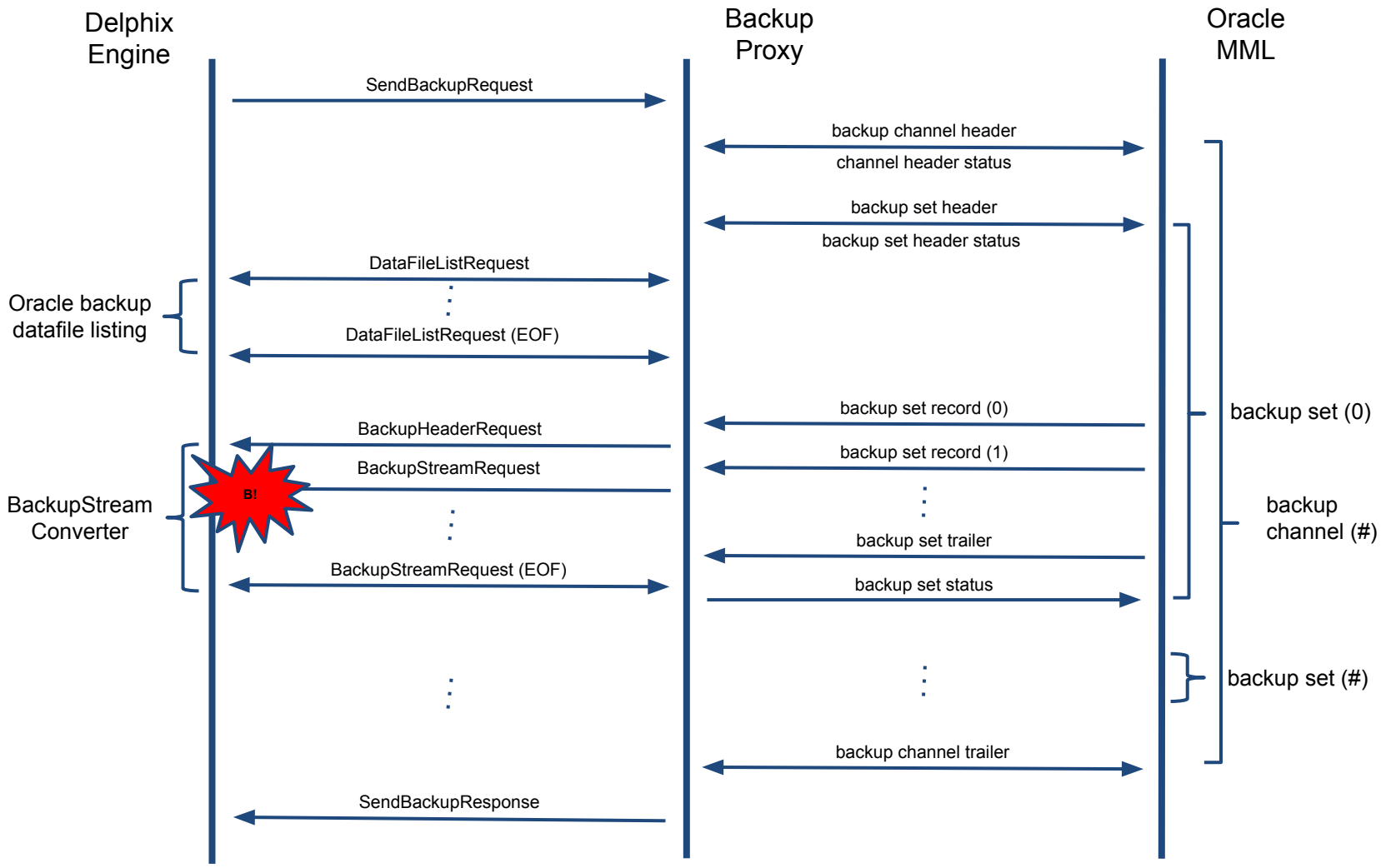
Asynchronous Support in DSP

- Service interfaces for asynchronous execution

DSP	Relation	Java
ServiceRequest ServiceResponse	≈	Callable
ServiceFuture	**extends**	Future
ServiceExecutor	≈	ExecutorService

- Core asynchronous support extensions
  - Improved task management semantics
  - Multiple asynchronous task tracking
  - Exception stitching

A Real World Example (SnapSync)



The Diagnosis

- The backup execution topology

  D => P -> { { C# => { { d, d, … } { d, d, … } … } } { C# => … } … }

  where

  - D represents Delphix
  - P for Backup Proxy
  - C# for the backup channel
  - d each backup data record
  - => asynchronous execution over DSP, and,
  - -> local asynchronous execution

- The diagnosis when exception occurs while processing d

  D => P -> C# => d

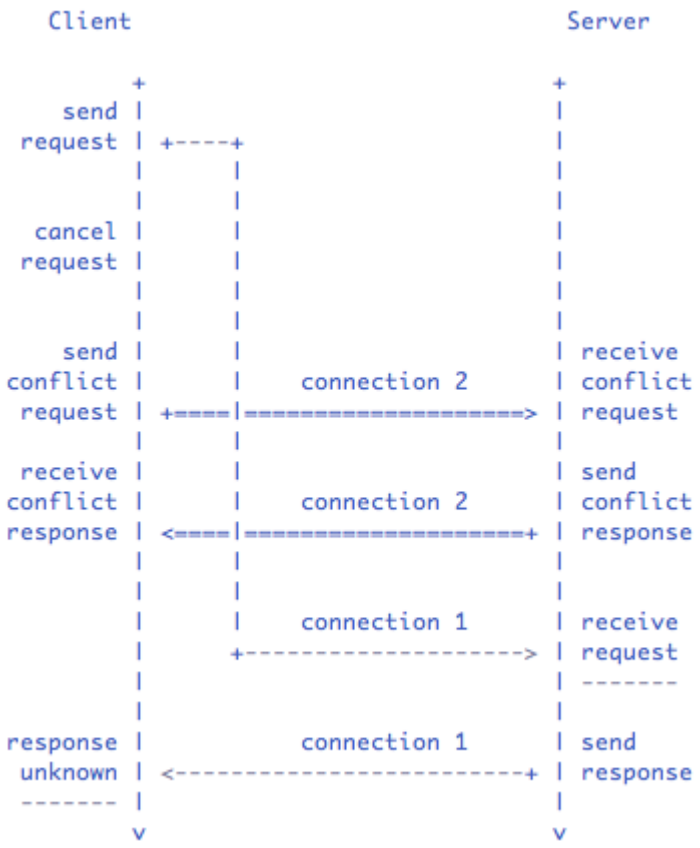
Infrastructure Exceptions

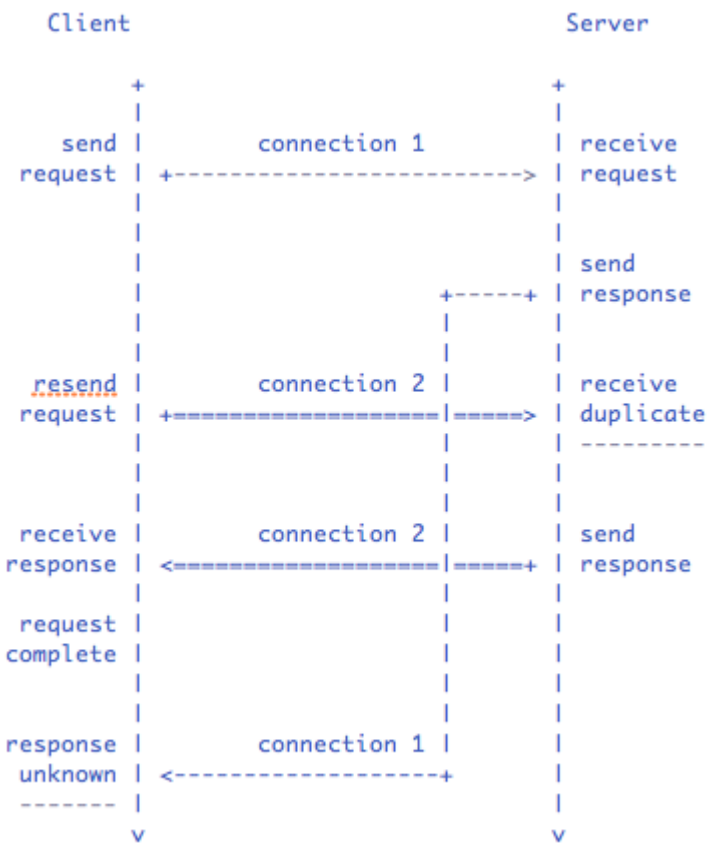
- Infrastructure exceptions lead to consistency violation.
- Consistency means agreement on the task execution state
- Consistency is implied in the local case
  - The callee is always reachable by the caller and vice versa
  - The callee cannot be terminated independent of the caller and vice versa
  - The data passed between the caller and callee is never corrupted
- Consistency is hard to achieve in the distributed case

Common Causes for Consistency Violation

- Connection reset
  - Is the task lost on its way to the server
  - Is the response lost on its way back
  - Is the task or response still in transit
- Data integrity
  - Is the task really the same as what's sent
- Task management
  - Is the task still active on the server
  - Could the task be activated in the future
- Client restart
  - Is there task remaining from previous incarnation of the same client
- Server restart
  - Is the task executed prior to server restart

Consistency Illustrated





DSP Consistency Semantics

- Task execution
  - The task guaranteed to be processed on the server
  - The task not processed more than once on the server
  - At least once semantics in case of server restart
- Task termination
  - The task, if active, quiesced on server
  - The task not activated again on server
  - The response never to be seen again on client
  - Eventual consistency guarantee in case connectivity loss
    - Availability chosen over consistency in case of partition (CAP)
    - Consistency ensured during session continuation before allowing application to continue

Recovery Strategy

Failure	Action	Impact
Connection Loss	Fail All Outstanding Commands on the Connection and Retransmit Over Another	None
Data Corruption	Sever the connection to force connection recovery	None
Connectivity Loss	Queue All Outstanding Commands in the Session for Recovery	None
Connectivity Restore	Session Continuation, Outstanding Command Recovery, and Aborted Command Quiescing	None
Client Session Loss	Session Reinstatement	None
Server Session Loss	Session Continuation Failure - Abandon Client Session	**Application Notified**
Command Abort	Task Management Over Command Connection, or; Task Management Over Working Connection, or;	None
	Task Management Upon Session Continuation	**Eventual Consistency**

Summary

- Exception classification
- Distributed diagnosis and programming abstractions
- Failure modes and consistency semantics
- How DSP plays into all this

Q&A

- Thank you for coming!