Dependability in Web Services

Christian Mikalsen

chrismi@ifi.uio.no

INF5360, Spring 2008

Agenda

Introduction to Web Services.

Extensible Web Services Architecture for Notification in Large-Scale Systems.
Krzysztof Ostrowski; Ken Birman. Web Services, 2006. ICWS ‘06. Sept. 2006 pp. 383 – 392.

Fault Tolerance Connectors for Unreliable Web Services.
Salatge, Nicolas; Fabre, Jean-Charles. Dependable Systems and Networks, 2007. DSN ‘07. June 2007 pp. 51 – 60.

Introduction to Web services

- Commonly used to create applications using Service Oriented Architectures (*SOA*).
- Loosely coupled services, communicating over HTTP.
- *WSDL* defines a contractual relation between a client and a web service provider.
- Web services often use *SOAP* to format request and replies.
- Due to the nature of the web, web services used in an application can be moved, deleted, unavailable due to communication failures or subject to other failures.

Introduction to Web services

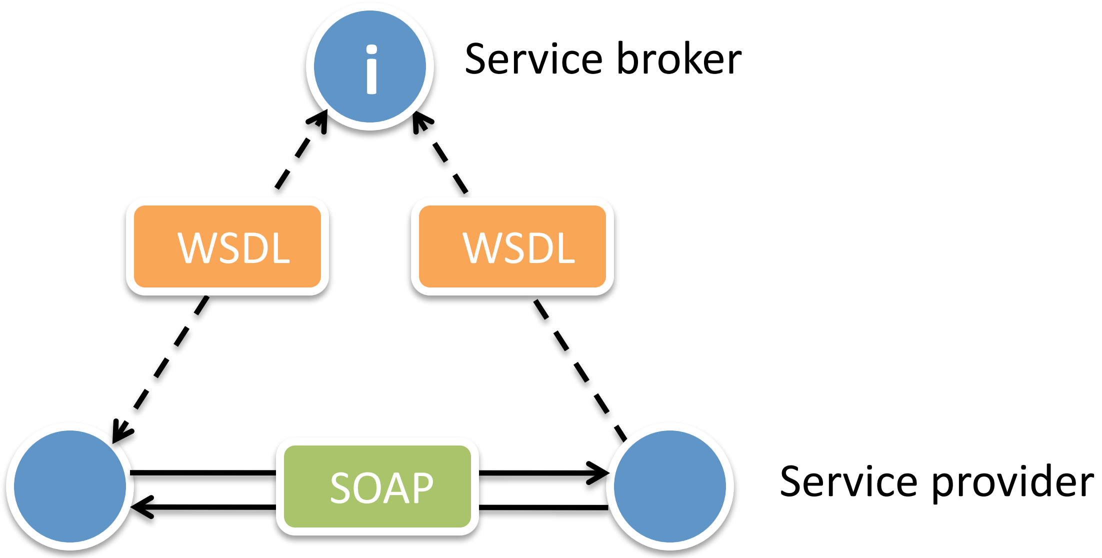


Figure based on illustration from Wikipedia (Web service article).

Overview

Extensible Web Services Architecture for Notification in Large-Scale Systems.

- Motivation
- Design principles and basic concepts
  - Scopes and the Scope Manager (SM)
  - Policies
  - Communication channels
  - Sessions
- Dissemination framework
- Reliability framework

Notifications

- Notifications are a widely-used primitive, allowing event driven programming and message exchange between Web Services.
- Standardized in WS-Notifications and WS-Eventing.
- Paper claims these standards are subject to limitations:
  - **Not self-organizing**
    Notification trees must be manually configured.
  - **Inability to use external multicast framework**
    Recipients must set up communication endpoints themselves.
  - **No forwarding among recipients**
    Recipients are passive, and unable to participate in distribution.
  - **Difficult to manage**
    Hard to support on Internet scale.
  - **Weak reliability**
    Limited to per-link guarantees (from TCP), no support for virtual synchrony or transactional replication.

Contribution

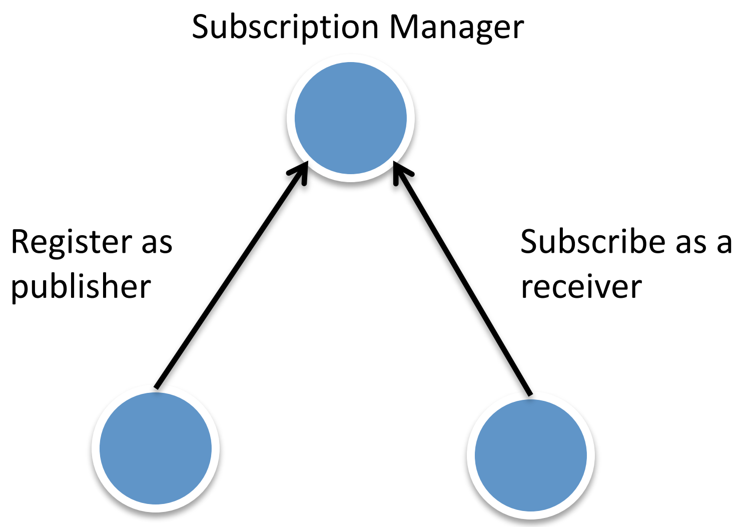
The paper claims to offer an approach to building large-scale systems for web services notification, free from the previous limitations.

Design principles

- **Programmable nodes**
  Nodes should be able to perform certain basic operations.
- **External control**
  Nodes should be controlled by trusted external entity.
- **Hierarchical structure**
  Messages are delivered reflecting policies at different levels.
- **Isolation and local autonomy**
  Freedom in how messages are forwarded internally in administrative domains.
- **Channel negotiation**
  Channel creation should allow handshake – agree on configuration.
- **Managed channels**
  Active contracts – recipients influence how senders are sending.
- **Reusability**
  Policies for message forwarding should be reusable.

Basic concepts (I)

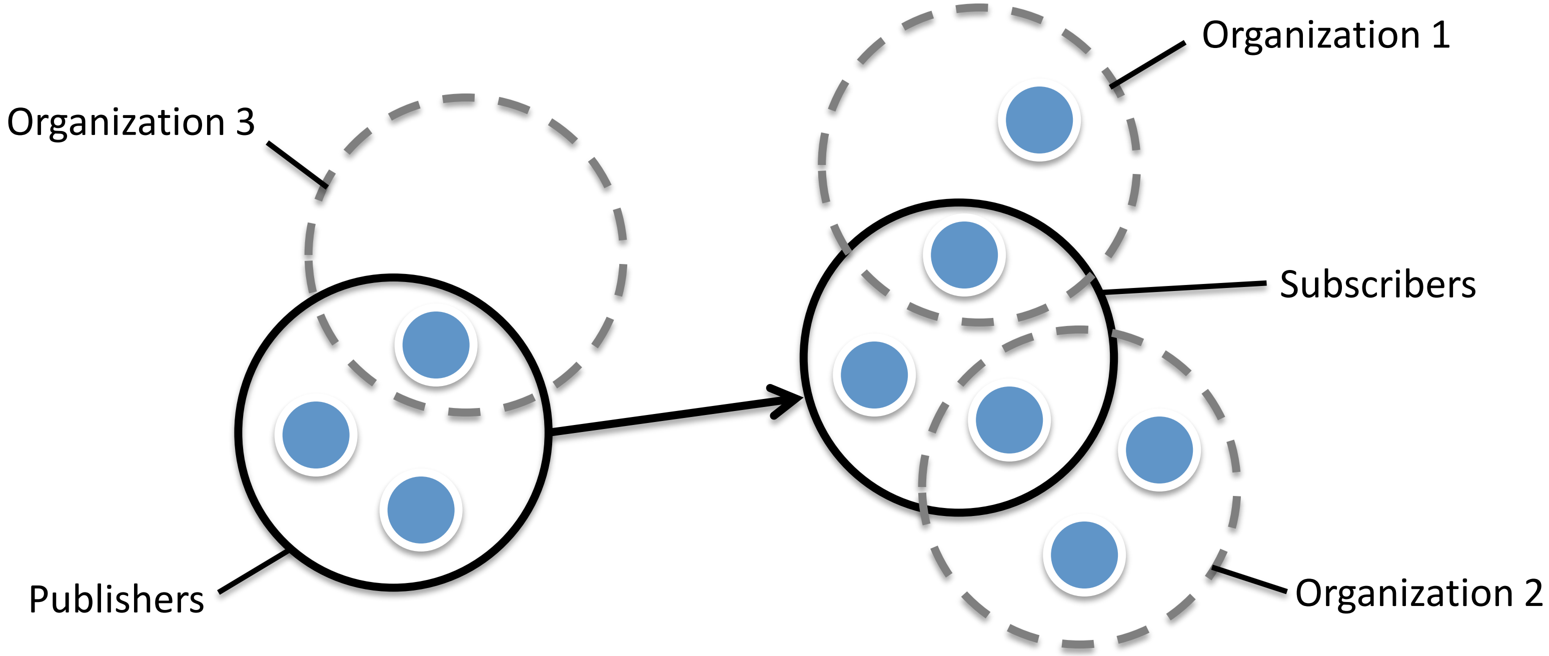
- Notifications are associated with *topics*, produced by *publishers* and delivered to *subscribers*.
- Prospective publishers and subscribers register with a Subscription Manager, which can be independent.



We may have more than one publisher for any topic.

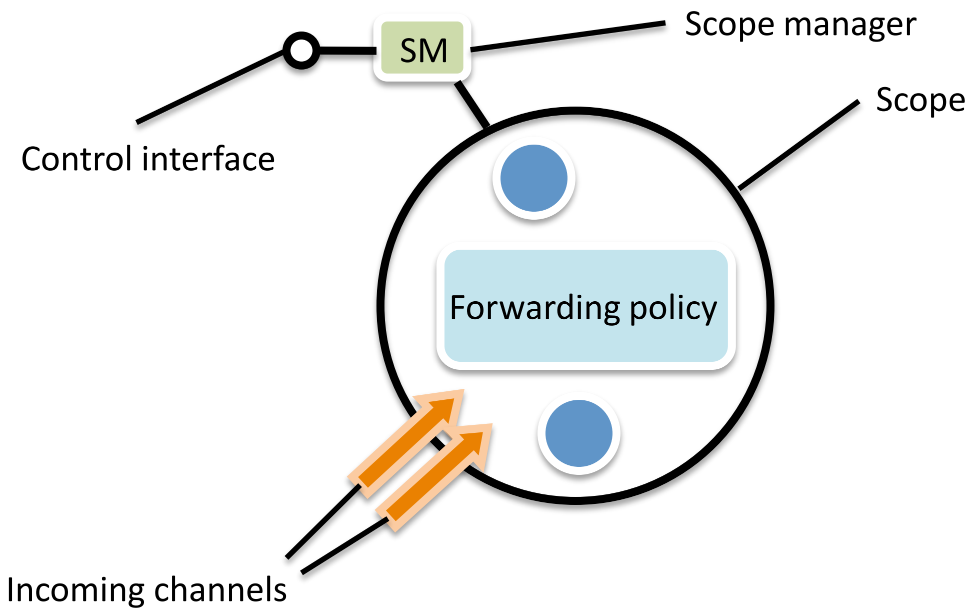
Basic concepts (II)

- Nodes reside in *administrative domains*, in which they are jointly managed.
- Publishers and subscribers may be scattered across many different administrative domains, but they must cooperate in delivery of messages.



Scopes

- A scope represents a set of jointly managed nodes.
- A scope may be a single node, all nodes within an AD or a set of nodes clustered by another criteria.



Scopes

- Often, a scope for example may be a LAN, but such one-to-one relationship not assumed.
- A scope requires infrastructure to maintain membership and administration.

- The *span of a scope* refers to the set of all nodes at the bottom of the *hierarchy of scopes* rooted at that scope.
- Publishing a message consists of delivering it to all subscribers in the span of some global scope.

Hierarchical view of the network

- The basis for the system is a hierarchical view of the network.
- Subscribers of a topic are divided into subsets, each belonging to an administrative domain. Continuing this recursively, we get a hierarchical structure.
- Each administrative domain is responsible for managing registration of its own publishers and subscribers.
- Scalability arises by *divide and conquer*.

The Scope Manager

- The Scope Manager provides infrastructure for managing membership and administration for one or more scopes.
- The Scope Manager offers a control interface, which is a Web Service hosted at a well-known location.
- The control Web Service dispatches control requests to the scope(s) it controls.
- The Scope Manager signals scope members to create *channels* and *filters* when membership or subscription changes.

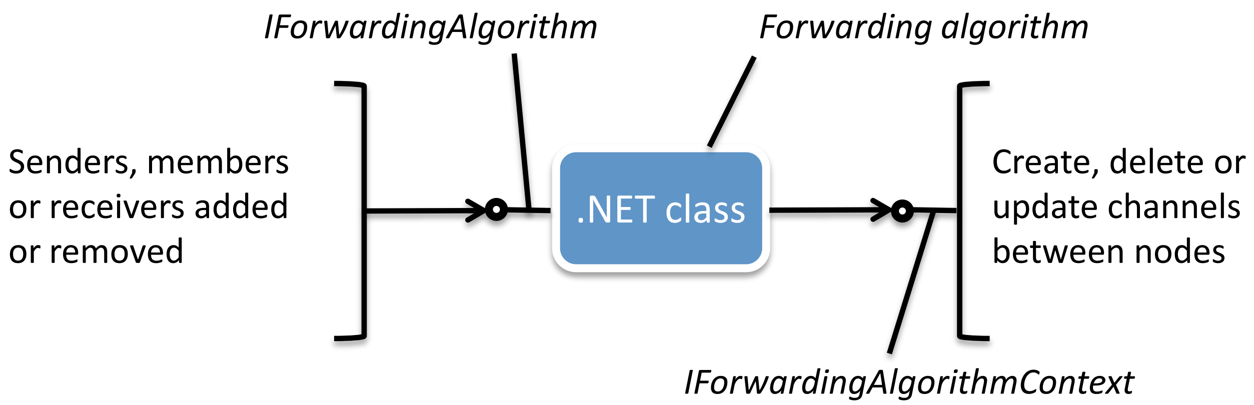
Policies (I)

- Each scope is configured with a policy defining how messages are forwarded among its members and to sub-scopes (on a per-topic basis).
- Policies are always defined at the granularity of X’s members (not individual nodes).
- A policy may require sub-scopes to perform specific forwarding, but the sub-scope may perform this internally as it wishes.

- This forwarding structure completely determines the way messages are forwarded.

Policies (II)

- A policy is an algorithm that lives in an abstract context, with a fixed set of events, operations and attributes.
- Prototype implementation with policies as .NET classes, stored on a library server.
- The scope manager maintains mappings from topics to policies – a graph of channels and filters to apply to them.



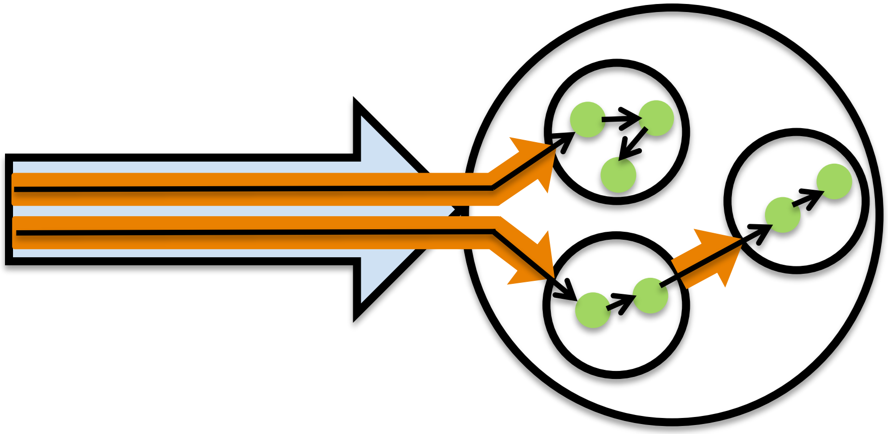
IForwardingAlgorithm
Forwarding algorithm
Senders, members or receivers added or removed
.NET class
Create, delete or update channels between nodes
IForwardingAlgorithmContext

Communication Channels (I)

- Channel is a mechanism through which messages can be delivered to all the nodes in the span of a scope, subscribed to a set of topics.
- Connecting scopes asks SM for a specification of the channel to be used for messages of topic T.
- Is either an address/protocol pair, a reference to an external multicast mechanism, or a set of sub-channels with accompanying filters connecting to other scopes.
- *Filters* control which messages are forwarded over a channel, and can optionally tag messages.

Communication Channels (II)

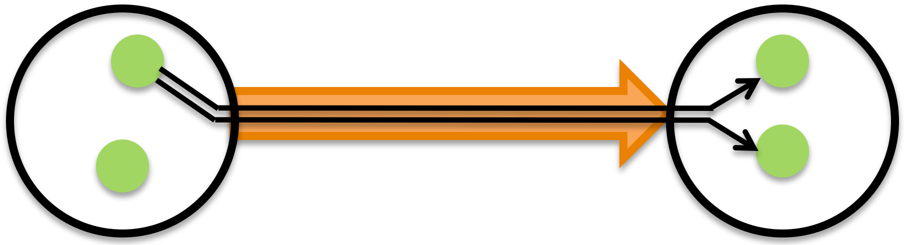
- For a single node scope, a channel might be a address/protocol pair, which a local process would use to open a socket.
- For scopes with several nodes, a channel may be a multicast address, to which the member nodes listen.
- In an overlay network, channels may lead to nodes that can forward the message across the overlay.



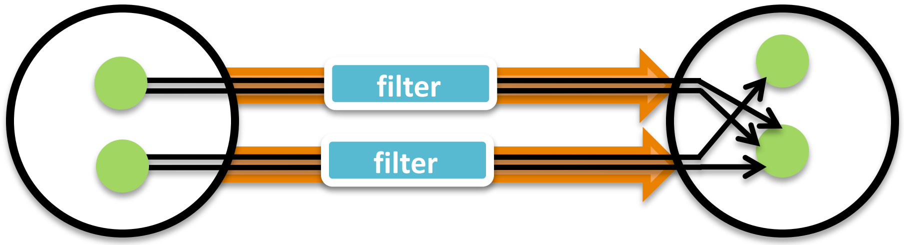
Communication Channels (III)

- Scopes spanning over several nodes are called *distributed scopes*, and as such cannot directly send messages or execute filters.
- Delegation
  - Rely on the fact that if a scope receives a message for a topic, then some of its members must receive them.
  - When using delegation, the scope requests that one such sub-scope create the channel on behalf of it. Done recursively, down to the level where a single node is requested to create the channel.
- Replication
  - In the case of replication, the scope requests several of its sub-scopes to create the channel, but constraints the channels with a filter implementing round robin operation.

Communication Channels (IV)



Delegation



Replication

Reliability (I)

- Reliability is provided by hierarchical *recovery scopes*, similar to dissemination scopes.
- The separation of dissemination/recovery makes it possible to combine an arbitrary unreliable notification mechanism with a wide range of reliability protocols.
- Reliability scopes are also controlled by a Scope Manager, typically the same as the dissemination scope.

Reliability (II)

- Recovery in a scope is modeled as recovering within sub-scopes, and then among the sub-scopes.
- A *recovery domain* is created to handle loss recovery and other reliability tasks for a specific set of topics in a scope.
- The recovery domain has a recovery algorithm, specifying how members of the recovery domain should exchange state and forward lost messages to each other.
- A recovery domain may contain sub-domains handling recovery for a set of subscribers in a sub-scope.

Reliability (III)

- Combined, we get a complete recovery structure:
  - The recovery domain for a scope specifies how recovery is handled in the specific scope.
  - The super-scope uses recovery information from all its sub-scopes, and combines them into its own recovery domain.
  - This continues hierarchically, until we have a complete recovery structure.
- Recovery domains actually handle recovery for *sessions*, not just for specific topics.

Epochs

- Epochs correspond to *membership views*, in group communication.
- When the set of subscribers change, a new epoch is started. During an epoch, membership is constant.

- **All reliability guarantees are expressed in terms of epochs.**

Sessions

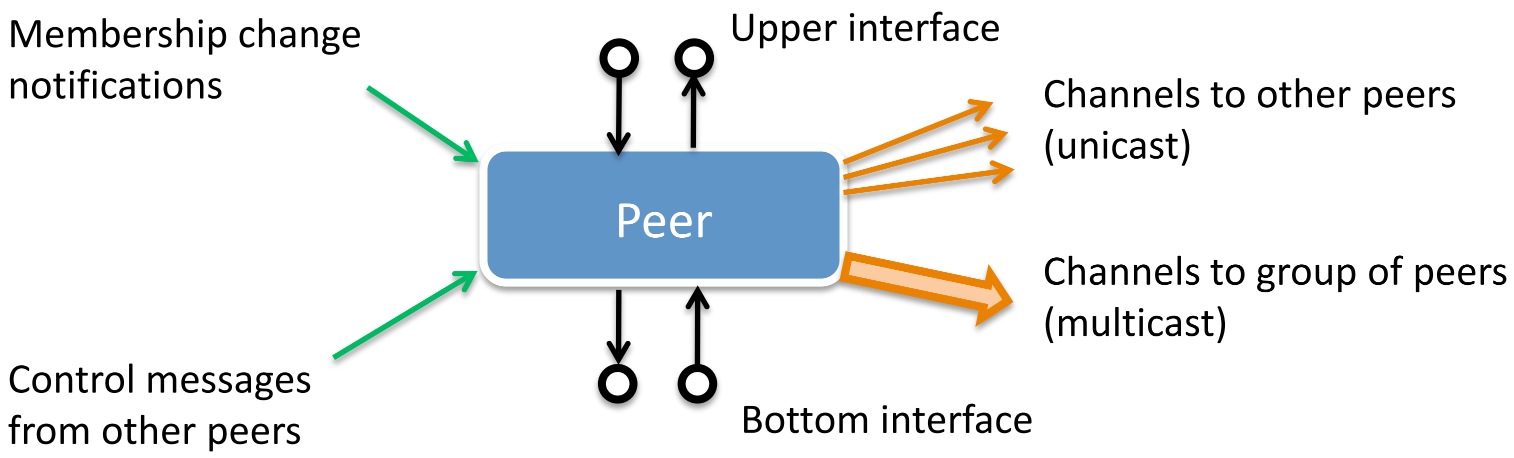
- Sessions are a similar to epochs, but a new session may be initiated even if membership is unchanged.
- Subscribers are notified of beginning and endings of sessions.
- Sessions are given a session number by the top-level SM, and published messages are tagged with the most recent session number.
- No nodes process messages received in a session until it is notified that it should accept messages of the session.
- When a new session is started, no messages are sent in the old session, which finally completes flushing, cleanup and other reliability mechanisms.

Modeling recovery algorithms

- The recovery framework is based on the abstract model of distributed protocols dealing with recovery and other reliability properties.
- Different recovery protocol can be defined in terms of a group of peers cooperating by sending control messages and forwarding lost packet to each other.
- Peers are designed with abstract interfaces to allow for a wide range of common recovery protocols to be implemented and used.
- Paper discusses detailed implementation of Reliable Multicast Transport Protocol (RMTP).

Peers

- Peers have an *upper interface* for communication with a *controller* (for aggregate operations, retransmission to all nodes etc.)
- Peers have a *bottom interface* to change and inspect local state (inspecting message order, marking messages as deliverable etc).



Membership change notifications
Upper interface
Channels to other peers (unicast)
Peer
Channels to group of peers (multicast)
Control messages from other peers
Bottom interface

Evaluation

- The paper claims that the strength of the design is its extensibility, ability to use a wide range of transport and recovery protocols, and enabling global publish-subscribe cooperation among independent parties.
- “Such benefits are hard to quantify. However, in certain scenarios, our approach also greatly improve scalability”.
- The performance of the approach is not evaluated in the paper.
- The authors are in the process of creating a reference implementation of the infrastructure of the paper, which should lead to specification like WS-Notification.

Conclusion

- Existing Web Services notification and eventing standards are useful for many applications, but have serious limitations for large-scale deployment.
- The paper proposes a design free from these limitations, using principles of hierarchy and local autonomy.
- The approach is extensible, has ability to accommodate a wide range of protocols for dissemination and recovery, and can assist in setting up a global infrastructure.
- However, performance has not been extensively been analyzed and a reference implementation is being implemented at the time of writing.

Overview

Fault Tolerance Connectors for Unreliable Web Services.

- Motivation
- Proposed solution
  - Concepts
  - Architecture
- Case study and experiments
- Conclusion

Motivation

- Web Services are now common, and more critical applications will use WS in the future.
- Applications with high dependability requirements may be composed a combination of highly reliable Web Services. However, such highly reliable services may be difficult to find.
- Highly reliable WS can be specifically developed, but that is contrary to the philosophy of SOA.
- New Web Services are typically made by combining existing unreliable services.
- We are therefore interested in an approach where we can equip existing, unreliable Web Services with additional fault tolerance mechanisms.

Contribution

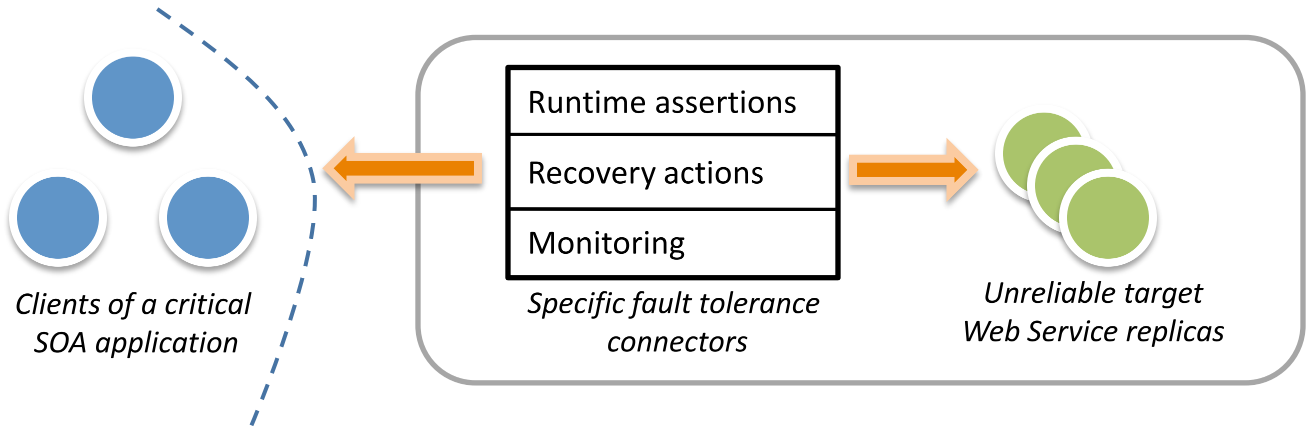
The paper outlines a solution using customizable *fault-tolerance connectors* to add fault-tolerance to existing, unreliable Web Services.

Connectors (I)

- Connectors are software components that are able to capture Web Service interactions, and perform fault-tolerance actions.
- Connectors can be designed by clients, providers or reliability experts using the original WSDL description of the service.
- The connectors insert detection actions (“runtime assertions”) and recovery mechanisms (based on various replication strategies).

Connectors (II)

- Connectors provide various mechanisms:
  - **User-defined runtime assertions**
    Applying checks to input/output requests, with error detection.
  - **Recovery actions**
    Recovery actions based on different replication models can be applied, depending on the target Web Service.
  - **Monitoring and error diagnosis**
    Collecting error information, leading to extended error reports.

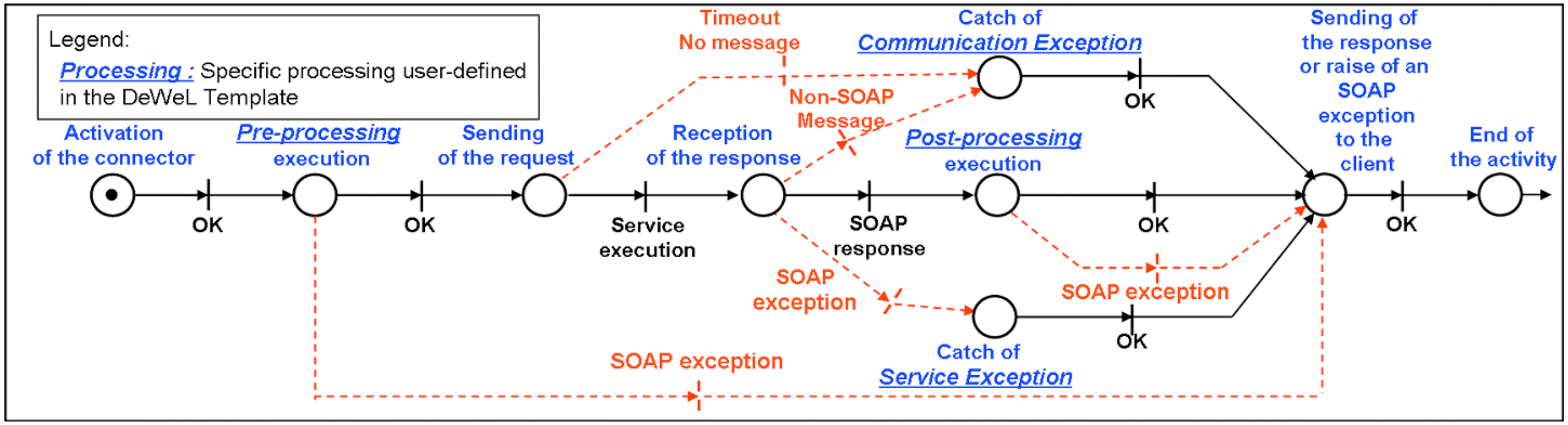

Runtime assertions
Recovery actions
Monitoring
Clients of a critical SOA application
Specific fault tolerance connectors
Unreliable target Web Service replicas

Connectors (III)

- Connectors must be highly reliable software components.
- A specific language, DeWeL, was developed to prevent software faults using compile-time and runtime verification when creating connectors.
- Applies common strict coding standards, such as no dynamic memory allocation, no loops, no recursions etc.
- The result is a custom, reliable WSDL contract for the target Web Service.

Execution model

- An execution model describes the behavior at runtime:
  - Pre- and post-processing correspond to DeWeL assertions.
  - *RecoveryStrategy* parameterized with location of WS replicas.
  - Predefined *CommunicationException* and *ServiceException*.


Legend:
Processing : Specific processing user-defined in the DeWeL Template
Activation of the connector
OK
Pre-processing execution
OK
Sending of the request
Service execution
Timeout No message
Reception of the response
Non-SOAP Message
Catch of Communication Exception
OK
SOAP response
Post-processing execution
OK
SOAP exception
SOAP exception
Catch of Service Exception
OK
SOAP exception
Sending of the response or raise of an SOAP exception to the client
OK
End of the activity

IWSD Platform

- The solution relies on a third-party infrastructure/platform between clients and WS providers.
- This platform, *Infrastructure for Web Services Dependability* (IWSD), offers functionality for loading connectors and running them.
- The platform must be highly reliable, achieved using other techniques.

Recovery and replication

- Recovery mechanisms rely on service replicas being available on the Net.
- *Identical services* are services with identical WSDL documents, but with another access point.
  - For example, Amazon has several replicas available across the world for its Web Services.
- *Equivalent services* are services that provide similar, but not identical, service. Allows us to fulfill a similar specification, or offer a degraded version of the service.
  - For example, consider an e-commerce site. If we are unable to perform a payment transaction with our typical payment partner, we can instead perform the transaction with an alternative partner.

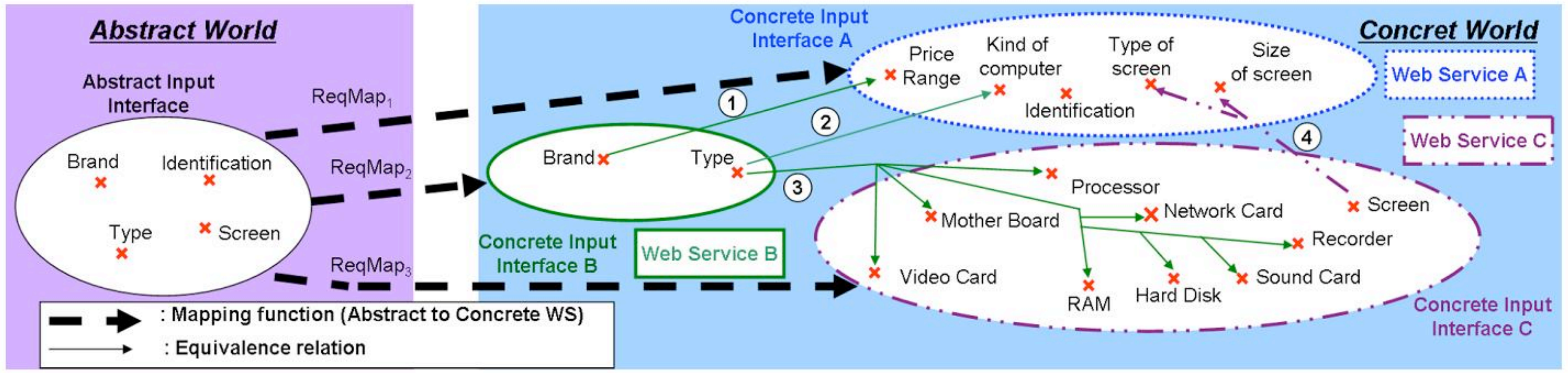
Equivalent services

- The framework uses the notion of *Abstract Web Services* (AWS), which is an abstraction of several similar services, with its own WSDL contract.
- The connector must convert between requests and replies, transforming them from abstract to concrete (and vice versa).

- *Equivalence relations* are semantic relations between two sets of parameters:
  Although syntactically different, sets of parameters or return values can describe the same information.
- The goal is to automate creation of the abstract operations from equivalence relations provided by the user (in a tool).

Equivalent services (example)

- Three equivalent Web Services (A, B, C), offering a client to purchase a computer.
- Task is creating the *minimal abstract interface* that allows us to map to the three concrete services.



- Service B can be called directly.
- Service A can be called using equivalence relations 1, 2 and 4 to generate input data.
- Service C can be called using equivalence relation 3 to generate input data.

Recovery strategies

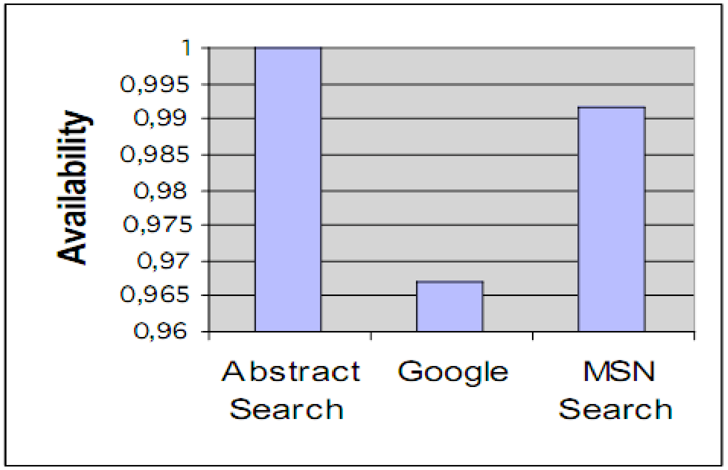
- Passive replication
  - Only one replica processes an input request.
  - In the case of failure, the request is forwarded to a spare replica.
  - Connector provides failure detection (communication failure, post-processing assertion errors etc.) and routing to spare replica.
- Active replication
  - The connector multicasts the request to WS replicas.
  - The connector either transmits the first response received, or uses a voting algorithm to tolerate faults in values.

Stateful services

- To support stateful services, we must also ensure state is managed between replicas. The paper offers two main execution models for handling state.
- The first approach is the *StatefulExecution* model:
  - The target Web Service must provide *SaveState* and *RestoreState* operations.
  - It is then up to the provider to handle the state complexity.
- Another approach is the *LogBasedReplication*:
  - The connector provides *StartSession* and *EndSession* operations to trigger logging of input requests.
  - When an error is detected by the connector, the log of requests is replayed with the new replica.

Case study: Equivalent services

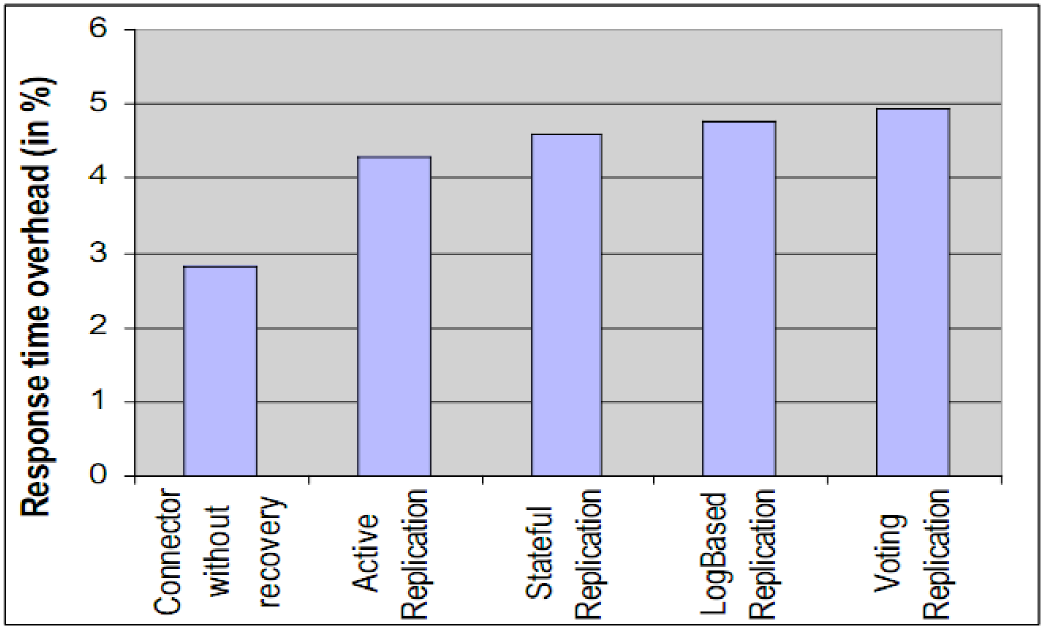
- Created an abstract Web Service providing search functionality, using Google and MSN Web Services as equivalent services.
- Used passive replication with Google as the primary service.



6% of request were redirected to MSN due to unavailability of Google WS.

Performance/overhead

- Experimentation for determining overhead of connectors compared to a direct client/provider connection.
- In the experiment, they report less than 3% overhead for stateless connectors without recovery.
- For stateful services, the number is still less than 5%.



Conclusion

- Service Oriented Architecture is useful in realizing large-scale applications, but multiple sources of failure can introduce faults and decrease availability.
- We want to make new services by combining existing services, and address/improve dependability by using fault-tolerance connectors.
- The fault tolerance connectors provide clear separation of concern between WS client and WS providers.
- *Identical* services found on the Net are useful, but even more useful is combining *similar* services transparently using equivalence relations, allowing us to take advantage of service/resource redundancy.

That’s it – questions?

Discussion topics

- How practical/feasible is the generation of equivalence relations in the real world?
- Performance of the reliable notification approach.
- Adaptability of the proposed approaches.