

Architectural Styles for Reliable and Manageable Web Services

Abdelkarim Erradi, Piyush Maheshwari
School of Computer Science and Engineering
University of New South Wales
Sydney NSW 2052, Australia
e-Mail: {aerradi, piyush}@cse.unsw.edu.au

Abstract

Service-Oriented Architecture (SOA) using Web services is considered as the state-of-the-art for distributed systems integration. However, despite the growing interest and the enormous potential of SOA, the uptake of this approach is still slow particularly for mission critical systems. This is partly because the focus is still on the definition and refinement of specifications rather than defining sound architecture styles and guiding principles. Building service-oriented applications is a complex undertaking, design patterns and frameworks have an important role to play to ease the process and to provide high Quality of Service (QoS) attributes. This paper discusses various Web services based integration patterns then it reports our experiments with wsBus, a lightweight service-oriented integration framework for dependable Web services interactions using a broker pattern. The main goal that guided the framework's design is to maximize decoupling complemented by scalable mediation service to cope with the inherent heterogeneity of systems involved. We discuss the system architecture and features then we report our experiences in building wsBus.

1. Introduction

The increasing move towards extensive automation and streamlining of business processes has fuelled interest in Service-Oriented Architecture (SOA) as business processes are likely to involve high number of participants with diverse and incompatible technologies. SOA is an approach for designing, architecting and delivering interoperability-centric and loosely coupled integration where software capabilities are seen as services delivered and consumed on demand. It promises applications that are composed of Internet accessible, independently developed and replaceable software components that are discovered at run-time [14]. SOA and Web services are being promoted as the foundation of next generation Business-to-Business integration (B2Bi) [10], wide-area Web-based Enterprise Application Integration (EAI), Peer-to-Peer (P2P) and Grid services [13]. The key goal of SOA is to provide a flexible and agile model of integration through reduced interdependencies as well as higher level abstraction and encapsulation, hence the increased ability to quickly align applications with changing business requirements. Message-based interactions using an orchestrated sequence of message exchanges (rather than invoking methods on interfaces) are seen as the core building block for

applications using this emerging computing paradigm as they provide the glue that ties services together [1]. However, the capabilities of current Web services frameworks are relatively limited compared to conventional middleware platforms like CORBA. Moreover, the focus is still on the definition and refinement of specifications rather than on the architecture.

Moving Web services beyond simple point-to-point applications to Internet-scale application-to-application integration, requires more than just a simple change of programming practices, rather a paradigm shift and mindset change is required to switch from RPC-based/object-based architecture to a loosely-coupled, message-focused and service-oriented architecture. A true SOA is realized when applications are built as self-contained, autonomous business services that interact by exchanging messages [16]. However, very few studies have closely looked at clearly defining such architecture or at evaluating and comparing the features set and the suitability of various architectural styles and design patterns to support SOA based applications.

This paper contributes to an improved understanding of Web services architecture. It briefly describes various Web services based integration patterns. It then presents wsBus, as a lightweight integration framework that mediates Web services interactions to enhance QoS while providing management capabilities to monitor the health, availability and performance of Web services. wsBus adds value and directly addresses some of the most critical issues surrounding the enterprise-grade deployment of Web services.

The rest of this paper is organized as follows. Section 2 discusses background material about Web services architecture and integration patterns while Section 3 describes the architecture and features of wsBus. Section 4 discusses related work and Section 5 finally concludes the paper and provides some thoughts on future work.

2. Background and problem area

Software architecture (SA) is the structure(s) of the system which comprise significant elements of the system, the externally visible properties of those elements (e.g. performance, reliability), and the relationships among them as well as the principles and guidelines governing their design and evolution over time [2]. The primary focus of SA is the decomposition, organization and interaction among architectural elements. SA raises the level of

abstraction and suppresses unnecessary details to provide a model of the system, which enables architects to focus on the analysis and the selection / tradeoff among competing design alternatives to achieve the desired quality attributes.

To better understand SOA we can apply Perry and Wolfe SA model [15] and we can define SOA as $SOA = \{Services, Messages, QoS/Policy\}$. With services being the components of the architecture and messages as the glue that holds the different pieces of the architecture together and the Quality of Service (QoS) attributes as the constraints on how the different services may interact and how they might be composed. In SOA, QoS refers to a wider range of non-functional service characteristics, such as availability, reliability, security, and performance.

Because of the lack of standard design and development techniques, architecting Web services based integration consists of defining a collection of views [8] and choosing/leveraging a set of well-known design patterns to describe the architecture. Design patterns are collections of structural and behavioral guidelines and rules for modeling frequently occurring design decisions [11].

2.1. Web services interaction patterns

Interactions between services can be achieved using various integration patterns such as point-to-point pattern, broker pattern and managed peer-to-peer (P2P) pattern [7].

The point-to-point pattern uses direct connection to integrate services. It is a simple integration pattern and requires minimal infrastructure using basic, widely supported standards and technologies, but capabilities such as reliability, security and management are left to the application developer. The number of point-to-point connections between n service providers and m service consumers is potentially up to $m(n^2)$, thus complicating manageability and maintainability as well as increasing operational cost as the number of connections grows. Moreover, the weakness of the P2P pattern is the duplication of transformation and routing logic between services, and the high configuration cost of endpoint address changes. To minimize these weaknesses, another layer of indirection (i.e., broker) could be added between endpoints.

The broker pattern and its variants provides a mediation layer with consistent handling of value-added QoS features by routing service traffic through a server to facilitate interactions through locating services, forwarding requests, and returning responses to clients. Clients access services by making service requests through the broker. A broker decouples interacting services by coordinating communication between endpoints and by assuming other responsibilities like validation of received message against the service's contract, access control to services, routing (i.e., determining the location of a target service), services registration (i.e., registering services to be discovered by other systems), transformation (i.e., conversion from one format to another), encryption, reliability of message

delivery and other QoS characteristics that can be centrally configured and enforced. The broker also bridges incompatible service policies (e.g., message compressed using Direct Internet Message Encapsulation (DIME) vs. message compressed using Message Transmission Optimization Mechanism (MTOM)).

The broker pattern provides the following benefits:

- Reduced coupling as the broker encapsulates access to the external resources. Applications that use a resource no longer require information about how to access that external resource.
- Improved modifiability as the interaction between services is mediated by a common controlled access point, for example a change in the location of a service is done once at the broker. In addition, the broker can implement some of the error handling that each application would otherwise have to perform.
- Improved security as the broker can be used to implement security policies and to bridge between different security mechanisms. It also allows metering access to the external resource as well as implementing business rules that control access.
- Improved interoperability as requesters using one protocol (e.g. SOAP-over-HTTP) can interact with Web services using different protocol (e.g., SOAP-over-JMS). Also filters can be used to validate or manipulate messages traveling through the broker.

However this pattern might reduce availability by introducing a single point of failure and congestion, thus requiring redundant/clustered configuration of the broker. The performance might also be reduced because of overheads and increased latency compared to direct communication.

The managed P2P combines characteristics from the previous two architectures. Web services interactions are channeled through local proxies while management data (e.g., response time) gathered during interactions are periodically dispatched to authorized subscribers (e.g., a central service management hub).

Taking into account the above-mentioned features of both models, the broker pattern is still more advantageous for the provision of large scale e-Services. Thus, it is the architecture used by our approach; future work will explore other interaction patterns and compares them.

Our experiments with wsBus reported in [5, 6] shows that the broker pattern provides higher Quality of Service (QoS) and more reliable inter-application messaging.

2.2. Motivation for wsBus

wsBus is an important architectural component for SOA-based applications. It is an enhanced service registry as well as a service intermediary that enhances and manages the delivery of Web Services by providing run-time support for reliable messaging, security, routing, monitoring and managing Web Services. It can be deployed as a client side

proxy, or as a server side Web services façade, or as a standalone broker. wsBus aims to facilitate large scale deployment of Web services in a secure, reliable and consistent manner by offering the following features:

- Enhanced service registry that can be used as a private UDDI with richer metadata descriptions including communication protocols, QoS capabilities and service rating as well as defining ‘communities’ of equivalent services. This feature is a step towards replaceable services. Because of the challenges presented by dynamic binding, we assume that after contractual relationships with service providers have been established and the behavior of the services involved has been tested, ‘compatible’ services will be registered and configured as a community in wsBus. At runtime wsBus will choose between equivalent services depending on the configuration supplied during registration as well as the QoS of prior interactions.
- Hide internal network details from service consumers: with wsBus acting as a proxy, when a service requests WSDL description of a registered service, wsBus automatically generate a re-factored WSDL based on the abstract description as the target Web service but with the protocol bindings requested by the client and with the endpoint pointing to the wsBus URI. This mechanism hides internal deployment details and increases flexibility. For example, it allows a service to be moved from one machine to another or to be deployed on multiple machines without affecting service consumers.
- Block unauthorized SOAP messages by authenticating requests and making sure that the source service is authorized to send messages to the destination service. This is achieved either by using the user id/password credentials contained in the SOAP message (as defined by WS-Security) or by authenticating the server that sent the message using its digital certificate.
- Multi-protocol services to bridge interfaces and protocol differences. This is motivated by the difficulty to dictate specific protocols and policies to business partners. Therefore, wsBus infrastructure acts as a service façade exposing services via number of protocols and policies that can be declaratively defined and enforced. For example, for secure access, a Web service client can choose between https and WS-Security over http.
- Load balancing for Web services by redirecting high load jobs or messages for platinum clients to special servers
- Improved error handling through configured re-routing or retry mechanisms
- Protect applications from emerging and rapidly evolving Web services standards: wsBus provides an extensible infrastructure to abstract away and implement standards compliance without application changes.
- Protect servers from overload by queuing or redirecting messages when the average response time goes beyond a certain threshold that is configured on a specific Web service using wsBus management console. Interested

parties can also subscribe to violation alert notifications by e-mail or SOAP action. For example, a trigger could be configured for an order processing service to raise an alert when more than 10 orders are being processed per second or if order processing takes more than 3 seconds. wsBus could then be configured to respond to the trigger by queuing orders or by processing high value orders first. This allows wsBus to perform basic load balancing and fail-over.

- Provide other added value services like Service Level Agreements (SLAs) monitoring, encryption, metering, and billing services

Our design of wsBus aims to enhance the reliability of Web service messaging, improve the scalability of its service provision, and maintain the security of all communication, whilst preserving the natural strength of Web services (i.e., interoperability). However, wsBus could also become a bottleneck but this can be addressed by clustered deployment.

It is also worth noting that we assume that Web services interactions channeled through wsBus are stateless, meaning that messages are self-contained with enough information (or links to persisted data) to allow the destination service to establish the message context. This assumption is fundamental to reliability and scalability properties of wsBus, it is also an encouraged practice in the Web services literature as in [1].

This paper concentrates on the architectural features of wsBus.

3. wsBus architecture and design

This section introduces the fundamental concepts and the main building blocks of wsBus.

wsBus uses broker and intercepting filters design patterns. Basically when a service is provided through wsBus, first it needs to be registered and configured (e.g., number of retries in case of failure, list of equivalent services known to wsBus, access control...etc). Service consumers need to sign up and create an account with wsBus then they can search for services provided via wsBus, download the WSDL of the selected service and start interacting with the desired service. As services are used various monitoring data get collected.

3.1. wsBus architecture

wsBus is a mediating component that provides a configurable framework for reliable Web services interactions. wsBus provides various channels (i.e., virtual endpoints) to access the registered Web services (each service is bound to one or more channels). The incoming message is assessed on arrival through the channel to determine the destination service. Filters can be configured to intercept and manipulate both request and response messages (e.g., transform old-format messages into new formats). The message is then passed through a reliability layer where it is checked -for expiration,

duplication, and ordering- and then it get queued for processing. wsBus then dispatches the message to the destination Web service and the response is passed back to

the requester via the same path. The diagram in Figure 1 shows wsBus architecture and its main components.

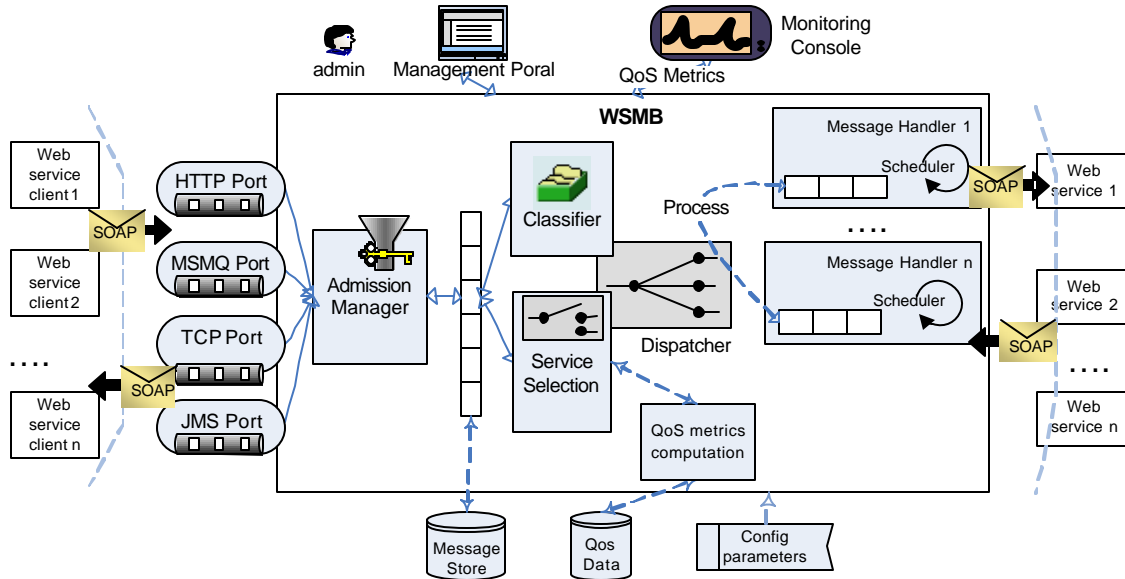


Figure 1. wsBus architecture and key components

The core idea of wsBus is to act as a bus which conveys SOAP messages from one end to another regardless the transport protocols (e.g., HTTP, JMS) being used in either ends.

wsBus infrastructure is based on interception. It sits between the transport and SOAP serialization/deserialization modules, inspecting messages and extracting/analyzing relevant SOAP headers (e.g., SOAP Action header) as appropriate and providing requested added-value services as instructed by the configuration settings.

A message store is used to enhance reliability by queuing and persisting messages. In case of a communication failure, the failing message is retrieved from the persistent storage and resent until an acknowledgement is received or the maximum number of retries is exhausted. The message store can also serve as a system log for debugging and auditing purposes. We are also planning to leverage the log for caching responses.

wsBus is also equipped with a user friendly Management Portal to register and configure Web services, and to define access control policy for the registered services.

wsBus Monitoring Console can be used to access QoS metrics and statistics gathered by the monitoring component during interactions as well as alerts raised in case of failures or violation of performance thresholds assigned to a registered service.

3.2. wsBus communication layer services

As shown in Figure 1, wsBus communication layer handles interception of messages and protocol bridging to allow interactions over multiple communication protocols (e.g., HTTP, JMS, WebsphereMQ, etc). The choice of the appropriate protocol to use can be decided based on the required QoS and available infrastructure. wsBus allows the Web service client to choose the desired transport without directly interfacing any deployed middleware that transports SOAP. From the schema of the endpoint URL of the destination Web service (e.g., jms:) wsBus transparently uses the appropriate communication channel. The aim here is to allow a client to transparently send a message to a queue as it would to an HTTP-based endpoint. wsBus demonstrates the ability to effectively switch between transports protocols by simply altering the target endpoint from a HTTP URL, to a listener waiting on JMS or WebsphereMQ queue in synchronous and asynchronous fashion.

Multi-protocol support virtualizes communication transports and helps to preserve interoperability while leveraging existing infrastructure. For instance, a purchase order encapsulated in a SOAP message could arrive using HTTP then wsBus could place the SOAP message in a message queue to be to be pickup, validated, prioritized and processed by an order processing workflow system. The requesting process will eventually receive a SOAP message confirming or rejecting the order through wsBus mediation. This allows shielding external users and partners from the internal implementation details while keeping data losses to a minimum.

3.3. wsBus message layer services

wsBus infrastructure relies on interception to process and relay messages (Figure 1). At start-up it loads the configuration and starts a pool of listeners to begin receiving messages. The listeners read messages from different channels and then call a series of configurable handlers to manipulate (pre/post-process) both request and response messages as instructed by the configuration settings. Messages travels along a configurable pipeline made of a series of handlers for processing and relaying messages. wsBus supports two types of handlers: built-in primitive and generic handlers (like validate a message against a schema, transform a message with XSLT, select a route using an XPath expression, split, log, encrypt/decrypt message, etc) as well as customized and specific handlers (like a custom handler to evaluate or enforce a business rule). Handlers can be assigned to different nodes in the infrastructure. Each node receives the messages into a private queue, and processes them through a configurable sequence of handlers, after that the processed message(s) are relayed to the destination(s) decided by the routing component (based on the destination specified in the message header, the configured routing rules or based on the content of the message body).

This architecture maximizes the system throughput by allowing parallelization of message processing tasks and the ability to deploy specialized resources for certain tasks like a server with more memory for validating large incoming XML messages. Moreover, the suggested architecture offers greater flexibility by allowing the messages processing logic to be easily adjusted by reconfiguration.

The messaging service component contains a number of sub-components through which an incoming message is required to undergo in order to be successfully processed:

- Listener: through which incoming messages arrive at wsBus and threads are created to handle these requests and direct messages further to the SOAP layer.
- Decryptor: is responsible for decrypting messages that were encrypted by the Web service client.
- Message backup: the message is then saved into persistent storage for future use (resend a message if the target Web service is not available, caching etc).
- Message classification and prioritization: As shown in figure 2, the classifier component assigns the incoming message an internal class-of-service depending on the rating of the requester (i.e., Gold clients, Silver clients, Bronze clients and best effort delivery clients) or the priority assigned to the requested service. Subsequently, the request is placed into a queue associated with its class-of-service. Then the scheduler dispatches the queued requests using the scheduling algorithm configured in wsBus. wsBus supports various scheduling algorithms for dispatching requests from queues like Round Robin (RR) algorithm and Weighted Round Robin (WRR) algorithm that assigns weight to queues to ensure that messages in higher priority

queues are fetched more often than those with lower priority [17].

- Message relay: The message relay component ensures that the invocation is done with maximum possibility of success. For example, if the target Web service is offline, then it will look up in wsBus database for a backup Web service that provides equivalent service. If found, the message will be sent to the selected Web service. If not, the message will be periodically resent until the target service responds, or the number of retries is exhausted or a timeout occurs.

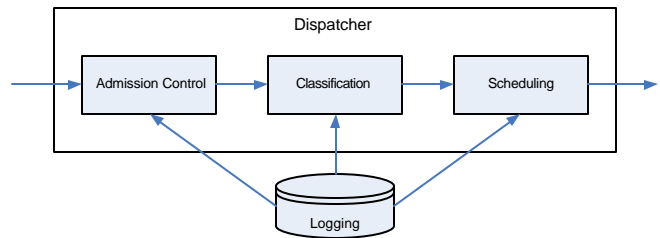


Figure 2. Dispatcher component

4. Related work and discussion

Several ongoing efforts recognize the need to extend the current technological infrastructure to foster wider adoption of Web services and SOA. In our early work [9], we tried to explore enhancing Web services by introducing Message Oriented Middleware (MOM) in the architecture. We came to the conclusion that it is unlikely that Web services can simply adapt existing MOM technologies to achieve higher QoS properties, given that they try to address interoperability on a scale at which conventional technologies have failed. Hence there is a clear need for new wide-area messaging infrastructure to enhance service delivery in Web-centric environment. wsBus is a first step towards achieving this. Its design was initially inspired from ideas introduced by the Web services Invocation Framework (WSIF) [12] but it adds more advanced QoS features as well as support for message-oriented interactions instead of RPC-based invocations promoted by WSIF. Web services are also an active area of standardization; our approach builds upon the building blocks of these standards and seeks to refine them by providing early implementations. The efforts around Enterprise Service Bus (ESB) [4] are closely related to our work but their focus is on EAI. In our approach we contribute to the realization of ESB vision while putting more focus on decentralized architecture, and broadening ESB application space as well as validating its feasibility and benefits.

Our future work enhance wsBus by extending some the ideas discussed in [3].

wsBus aims to extend and leverage the principles of messaging systems as well as the emerging Web services standards to support more reliable and manageable Web

services interactions using a decentralized architecture. The next stage of our research is to complete the development of the framework and to develop a number of services to fully test the features of wsBus. We are also planning to investigate possibilities to leverage some peer-to-peer techniques particularly for service discovery, enhanced scalability and exchange of QoS data between services (e.g., service rating).

5. Conclusion

This paper discussed various Web services interaction patterns, and then presented an initial design and prototype implementation of wsBus, a lightweight messaging framework using a broker pattern, it builds on current standards to meet the reliability and manageability requirements of Web services based integration. wsBus has the potential to grow as a reliable and QoS-aware integration infrastructure to move SOA beyond simple point-to-point service integration by facilitating large-scale implementation of the SOA principles with suitable service levels and manageability. The fundamental principle behind wsBus is abstracting away QoS features so they can be configured and enforced instead of being baked into the service logic. This results in a greater flexibility and a better maintainability and reuse. While not claiming to be a silver bullet, wsBus has achieved (in non-invasive way) some success in adding valuable QoS for Web services such as guaranteed delivery and recovery mechanisms, but the framework needs to be improved by adding support for transformation, content-based routing, caching and advanced QoS management services.

Acknowledgment

This work is part of a project jointly funded by the Australian Research Council and Microsoft Research Asia (ARC Linkage Project LP0453880). We thank Trung Nguyen Kien and Danny Choo for their contributions to the ongoing development of wsBus prototype.

References

- [1] Alonso, G., Casati, F., Kuno, H. & Machiraju, V. 2003, *Web Services: Concepts, Architectures, and Applications*, Springer Verlag, Berlin.
- [2] Bass, L., Clements, P. & Kazman, R. 2003, *Software Architecture in Practice*, (2nd edition), Addison-Wesley.
- [3] Birman, K., Van Renesse, R. & Vogels, W. 2004, 'Adding high availability and autonomic behavior to Web services', in *Proceedings of 26th International Conference on Software Engineering (ICSE 2004)*, pp. 17-26.
- [4] Chappell, D. 2004, *Enterprise Service Bus*, O'Reilly, ISBN 0-596-00675-6.
- [5] Erradi, A. & Maheshwari, P. 2005, 'wsBus: A Framework for Reliable Web Services Interactions', in *20th Annual ACM Symposium on Applied Computing*, Santa Fe, New Mexico.
- [6] Erradi, A. & Maheshwari, P. 2005, 'wsBus: QoS-aware Middleware for Reliable Web Services Interactions', in *IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05)*, Hong Kong.
- [7] Hohpe, G. 2004, *Integration Patterns*, Microsoft Press.
- [8] Kruchten, P. 1995, 'Architectural Blueprints – The “4+1” View Model of Software Architecture', *IEEE Software*, vol. 12, no. 6, pp. 42-50.
- [9] Maheshwari, P., Nguyen, T. & Erradi, A. 2004, 'QoS-based Message-Oriented Middleware for Web services', in *WISE 2004 Web Services Quality Workshop*, Brisbane, Australia, LNCS 3307, pp. 241-251.
- [10] Medjahed, B., Benatallah, B., Ngu, A. B. H. H. & Elmagarmid, A. K. 2003, 'Business-to-business interactions: issues and enabling technologies', *VLDB Journal - Springer*, vol. 12, no. 1, pp. 59-85.
- [11] Monday, P. B. 2003, *Web Services Patterns: Java Edition*, Apress.
- [12] Mukhi, N. K., Khalaf, R. & Fremantle, P. 2002, 'Multiprotocol web services for enterprises and the grid', in *Proceedings of the EuroWeb 2002 Conference on the Web and the Grid: From e-science to e-business*, Oxford, UK. <http://ewic.bcs.org/conferences/2002/euroweb/session5/paper2.pdf>
- [13] Open Grid Services Architecture Working Group (OGSA - WG). Available: <http://forge.gridforum.org/projects/ogsa-wg> [Accessed 30/08/2004].
- [14] Papazoglou, M. P. 2003, 'Service-Oriented Computing: Concepts, Characteristics and Directions', in *CAiSE 2003*, LNCS 2681 Springer 2003, Klagenfurt, Austria.
- [15] Perry, D. E. & Wolf, A. L. 1992, 'Foundations for the Study of Software Architecture', *ACM Software Engineering Notes*, vol. 17, no. 4, pp. 40-52.
- [16] Vogels, W. 2003, 'Web Services Are Not Distributed Objects', *IEEE Internet Computing*, vol. 7, no. 6, pp. 59-66. <http://weblogs.cs.cornell.edu/AllThingsDistributed/archives/000343.html>
- [17] Ye, N., Gel, E., Li, X., Farley, T. & Lai, Y.-C. 2005, 'Web-server QoS models: Applying scheduling rules from production planning', *Computers & Operations Research*, vol. 32, no. 5, pp. 1147-1164. http://ceaspub.eas.asu.edu/y/e/publications/v2/Ye_52.pdf