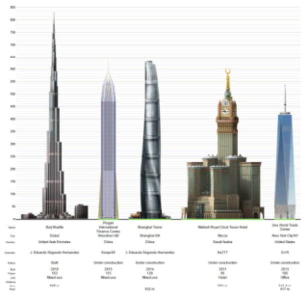
Script **generated by TTT**

Title:	Matthes: Soft-Arch (31.01.2012)
Date:	Tue Jan 31 18:11:58 CET 2012
Duration:	72:59 min
Pages:	60

Software Architectures

6. Architecture for Systems of Systems



Prof. Florian Matthes, Sascha Roth
Software engineering for business information systems (sebis)

wwwmatthes.in.tum.de

6 – Architectures for Systems of Systems

- "Conventional" Middleware for Distributed Information Systems [Al05]
  - RPC and Related Middleware
  - Object Brokers
  - Message-Oriented Middleware
  - EAI Middleware: Message Brokers
- Web Services
- Service-Oriented Architecture
- REST [Fi00]

Recommended Reading: [Al04]

Alonso and his co-authors deliberately take a step back. Based on their academic and industrial experience with middleware and enterprise application integration systems, they describe the fundamental concepts behind the notion of Web services and present them as the natural evolution of conventional middleware, necessary to meet the challenges of the Web and of B2B application integration.

From this perspective, it becomes clear why Web services are needed and how this technology addresses such needs.

Alonso, G.; Casati, F.; Kuno, H.; Machiraju, V.: *Web Services – Concepts, Architectures and Applications*. Springer. 2004

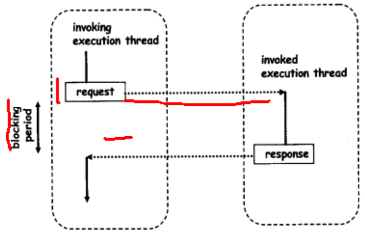
Rather than providing a reference guide or a manual on how to write a Web service, the authors discuss challenges and solutions that will remain relevant regardless of how emerging standards and technologies evolve.

Communication in an Information System

- The dominating characteristic of any software interaction is whether it is *synchronous* or *asynchronous*.
- Formally: blocking vs. non blocking
- Synchrony has nothing to do with concurrency and parallelism.

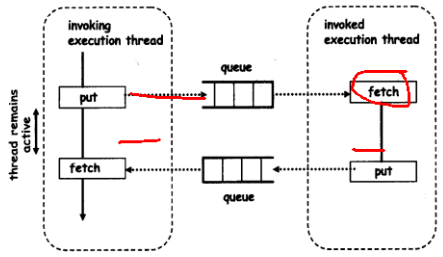
Synchronous or Blocking Calls

- In synchronous interaction, a thread of execution calling another thread must wait until the response comes back before it can proceed.
- This leads to simpler design which are easier to understand.
  - The state of the calling thread will not change before the response comes back.
  - There is a strong correlation between the code that makes the call and the code that deals with the response.
- Can be a significant waste of time and resources if the call takes time to complete



Asynchronous or Non Blocking Calls

- Simple example: e-mail
- A message is sent and, some time later, the program checks whether an answer has arrived.
- This allows the calling program to perform tasks in the meanwhile and eliminates the need for any coordination between both ends of the interaction.



Middleware

- Middleware offers programming abstractions that hide some of the complexity of building a distributed application.
  - The middleware takes care of some aspects.
  - The programmer has access to functionality that otherwise would have to be implemented from scratch.
- There is a complex software infrastructure that implements those abstractions.
  - Tends to have a large footprint
  - Extensions and enhancements of the original programming abstraction make the infrastructure more complex.

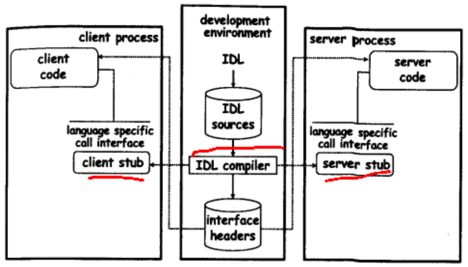
RPC and Related Middleware

- **Remote Procedure Call** (RPC) is the foundation underlying the vast majority of middleware platforms available today.
- Observation: procedure calls are a well-understood mechanism for transfer of control and data within a program running on a single computer.
- RPCs extend that mechanism to provide transfer of control and data across a communication network [BN84].
- RPC established the notion of **client** (the program that calls a remote procedure) and **server** (the program that implements the remote procedure being invoked).

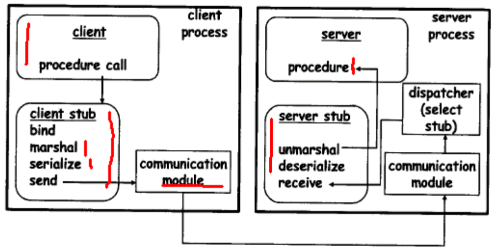
How RPC Works (1)

1. Define the interface for the procedure using an **interface definition language** (IDL).
   - Provides an abstract representation of the procedure in terms of what parameters it takes as input and what parameters it returns as a response.
2. Compile the IDL description, which produces:
   - Client stubs
     - When the client calls a remote procedure, the call that is actually executed is a local call to the procedure provided by the stub.
     - Locates the server (binding), formats the data (marshaling and serialization), communicates with the server, gets a response and forwards that as the return parameter.
     - Is a placeholder or proxy for the actual procedure implementation at the server.
     - Will be compiled and linked with the client code.
   - Server stubs
     - Will be compiled and linked with the server code.
     - Receives requests, deserializes and unmarshals the call, invokes the actual procedure implementation on the server and sends the result back to the client.
   - Code templates and references

How RPC Works (2)



How RPC Works (3)

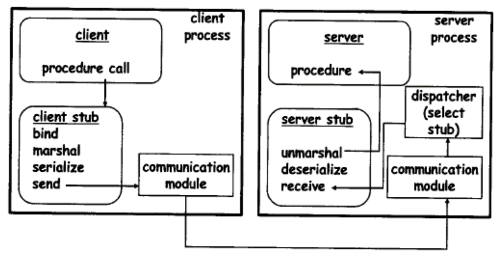


Binding in RPC

- In order for a client to make an RPC, it must first locate and bind to the server hosting the remote procedure.
- Binding is the process whereby the client creates a local association for (i.e. a **handle**) a given server in order to invoke a remote procedure.
- Binding can be either **static** or **dynamic**.

- In static binding, the client stub is hardcoded to already contain the handle of the server where the procedure resides (e.g., IP address and port number, Ethernet address, X500 address, …).
- Advantages of static binding: efficient and simple, no additional infrastructure is needed.
- Disadvantages: the client and the server become tightly coupled; if the server changes location the client has to recompiled with a new stub.
  - It is not possible to use redundant servers to increase performance; load balancing must take place at the time the clients are distributed and developed.

How RPC Works (3)



Binding in RPC

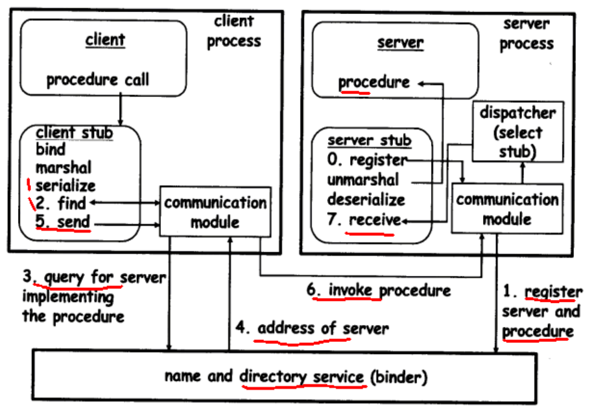
- In order for a client to make an RPC, it must first locate and bind to the server hosting the remote procedure.
- Binding is the process whereby the client creates a local association for (i.e. a **handle**) a given server in order to invoke a remote procedure.
- Binding can be either **static** or **dynamic**.

- In static binding, the client stub is hardcoded to already contain the handle of the server where the procedure resides (e.g., IP address and port number, Ethernet address, X500 address, …).
- Advantages of static binding: efficient and simple, no additional infrastructure is needed.
- Disadvantages: the client and the server become tightly coupled; if the server changes location the client has to recompiled with a new stub.
  - It is not possible to use redundant servers to increase performance; load balancing must take place at the time the clients are distributed and developed.

Dynamic Binding (1)

- Enables the client to use a specialized service to localize appropriate servers.
- Adds a layer of indirection to gain flexibility at the cost of performance.
- The **name and directory server** is responsible for resolving server addresses based on the signatures of the procedures being invoked.
- Enables dynamic load balancing.
- If the server changes location only the entry in the name and directory server has to be changed → **Decoupling** of client and server

Dynamic Binding (2)



RPC and Heterogeneity

- The stubs can be used to hide not only the distribution but also the **heterogeneity**.
- A naïve approach would use a different client and server stub set for every possible combination of platforms and languages -> 2**n***m* stubs for *n* client and *m* server platforms.
- More efficient: some form of **intermediate representation** so that client and server stubs only have to know how to translate to and from this intermediate representation (*n*+*m* stubs)
- The IDL
  - is used to define the mapping from concrete programming languages to the intermediate representation used in that particular RPC system,
  - serves for defining the intermediate representation for data exchanges between clients and servers,
  - defines how parameters should be represented and organized before being sent across the network.

RPC Design Decisions

Should RPC be **transparent** to the programmer?

- Pro: simplicity, programmers do not deal with distribution directly
- Contra: including a remote call changes the nature of the program deeply
  - Forcing programmers to use special constructions for RPC is a way if making aware of a remote call and the implications (performance, reliability)

Most modern RPC systems use a transparent approach.

The Eight Fallacies of Distributed Computing [De97]

Essentially everyone, when they first build a distributed application, makes the following eight assumptions:

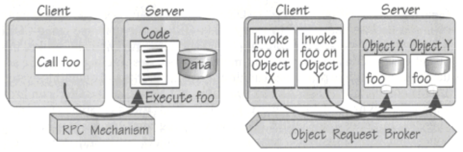
1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

All prove to be false in the long run and all cause big trouble and painful learning experiences.

- "Conventional" Middleware for Distributed Information Systems [Al05]
  - RPC and Related Middleware
  - Object Brokers
  - Message-Oriented Middleware
  - EAI Middleware: Message Brokers
- Web Services
- Service-Oriented Architecture
- REST [Fi00]

Object Brokers

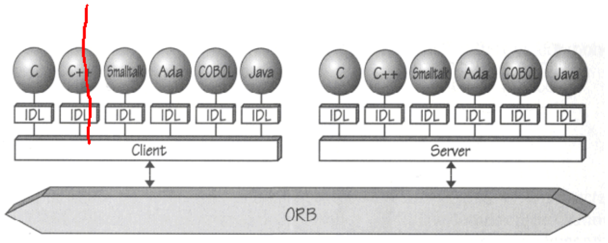
- Object broker extends the RPC paradigm to the object-oriented world; they provide services that simplify the development of **distributed object-oriented** applications.
- Difference to simple RPC: clients invoke a **method of an object**
  - Because of **inheritance** and **polymorphism** the function performed by the server object actually depends on the class to which the server object belongs.
  - The middleware has to bind clients with specific objects running on a server and manage the interactions between two objects.



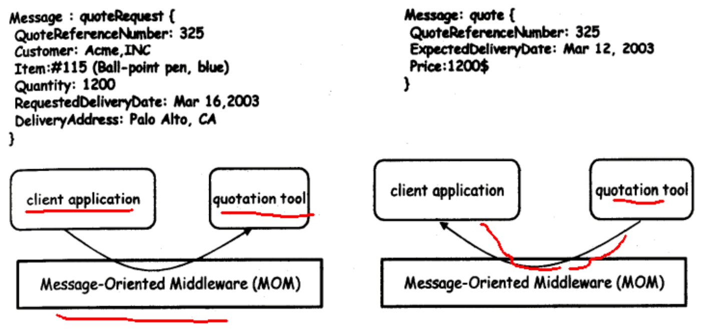
- With time, object brokers added features that went beyond basic interoperability, for example including location transparency, object lifecycle management, and persistence.

CORBA

- The best known example of object broker is the abstraction described in the **Common Object Request Broker Architecture** (CORBA) specification, which was developed in the early 1990s by the Object Management Group (OMG).
- It offers a **standardized specification** of an object broker rather than a concrete implementation.
- CORBA is agnostic with respect to both the programming language used to develop object-oriented applications and also the operating systems on which the applications run.



Message-based Interoperability (2)



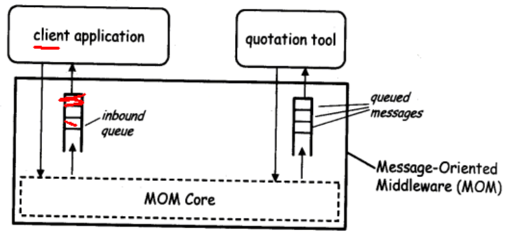
```
Message : quoteRequest {
 QuoteReferenceNumber: 325
 Customer: Acme,INC
 Item:#115 (Ball-point pen, blue)
 Quantity: 1200
 RequestedDeliveryDate: Mar 16,2003
 DeliveryAddress: Palo Alto, CA
}
```

```
Message: quote {
 QuoteReferenceNumber: 325
 ExpectedDeliveryDate: Mar 12, 2003
 Price:1200$
}
```

client application
quotation tool
Message-Oriented Middleware (MOM)
client application
quotation tool
Message-Oriented Middleware (MOM)

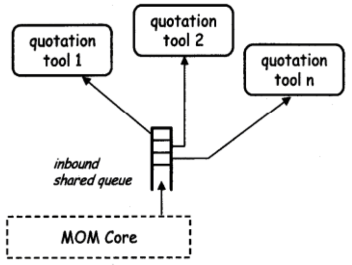
Message Queues

- One of the most important abstractions based on MOM is that of **message queuing**.
- In a message queuing model, messages sent by MOM clients are placed into a queue, typically identified by a name, and possibly bound to a specific intended recipient.
- Whenever the recipient is ready to process a new message, it invokes the suitable MOM function to retrieve the first message in the queue.



Shared Queues

- Shared Queues can be used to distribute load among multiple applications that provide the same service.
- The MOM system controls access to the queue, ensuring that a message is delivered to **only one** application.



Benefits of Message Queues

- It gives recipients control of **when to process** messages.
- Recipients do not have to be continuously listening for messages and process them right away, but can instead retrieve a new message only when they can or need to process it.
- Queuing is **more robust to failures** with respect to RPC or object brokers, as recipients do not need to be up and running when the message is sent.
- If an application is down or unable to receive messages, these will be stored in the application's queue (maintained by the MOM).
- Queued messages may have an associated **expiration date** or interval.
- Queues can be **shared** among multiple applications.

Interacting with a Message Queuing System

- Queuing systems provide an API that can be invoked to send messages or to wait for and receive messages.
- Sending a message is typically a non blocking operation.
- Receiving a message is instead often a blocking operation, where the receiver listens for messages and processes them as they arrive, typically by activating a new dedicated thread, while the main thread goes back to listen for the next message.
- Non blocking alternative: provide a callback function that is invoked each time a message arrives.

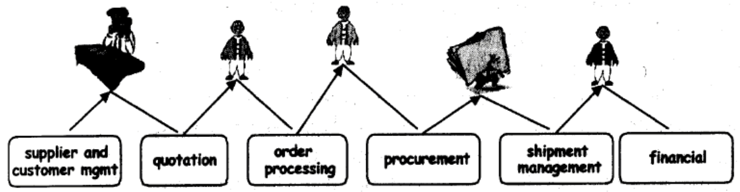
Java Message Service

- Java industry-standard API: Java Message Service (JMS)
- In JMS, a message is characterized by
  - a header, which includes metadata such as the message type, expiration date, and priority, …
  - a body, which includes the actual application-specific information that needs to be exchanged.
- Addressing is performed through queues:
  - senders (receivers) first bind to a queue, i.e., identify the queue to (from) which they want to send messages (receive messages from), based on the queue name,
  - then they can start sending (retrieving) messages to (from) the queue.
- JMS is simply an API and not a platform.
- JMS can be implemented as a stand-alone system or as a module within an application server.

From Middleware to Application Integration

- The use of middleware led to a further proliferation of services.
- Integration is now not only the integration of resource managers or servers, but also the integration of services.
- While for servers there has been a significant effort to standardize the interfaces of particular types of servers (e.g., databases), the same cannot be said of generic services.
- There was almost no infrastructure available that could help to **integrate services provided by different middlewares**.
- Middleware was originally intended as a way to integrate servers that reside in the resource management layer.
- → Enterprise Application Integration (EAI) appeared in response to this.
- EAI includes as building blocks the application logic layers of different middleware systems.

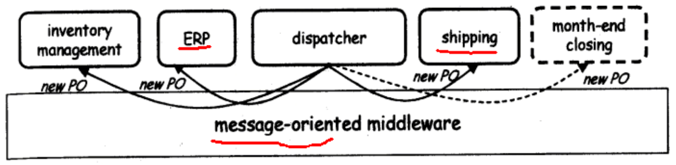
Example of Application Integration



- Different operating systems,
- Different interfaces (transactional/non-transactional, standard IDL/proprietary, …),
- Different data formats,
- Different security requirements (authentication), and
- Different middlewares

EAI Middleware: Message Brokers (1)

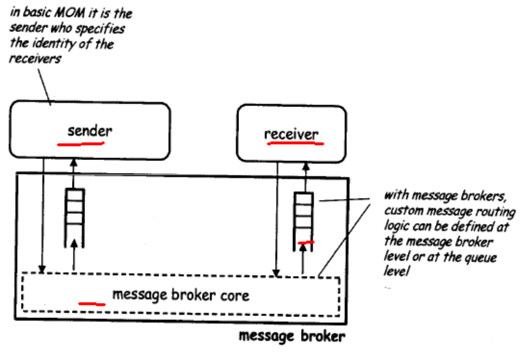
- Traditional RPC-based and MOM systems create **point-to-point** links between applications → they are rather static and inflexible with regard to the selection of queues to which messages are delivered.



- Message brokers **act as a broker** among system entities, thereby creating a (logical or physical) "**hub and spoke**" communication infrastructure for integrating applications.

EAI Middleware: Message Brokers (2)

sebis


in basic MOM it is the sender who specifies the identity of the receivers
sender
receiver
with message brokers, custom message routing logic can be defined at the message broker level or at the queue level
message broker core
message broker

Extending Basic MOM

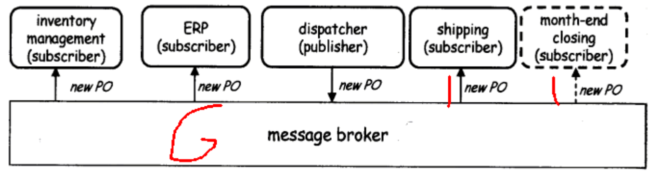
sebis

- Message brokers factor the message routing logic out of the senders and place it into the middleware.
- There is now a **single place** where we need to make changes when the routing logic for messages needs to be modified.
- **Routing logic**
  - can be based on the sender's identity, on the message type, or on the message content,
  - is typically defined in a rule-based language.
- Message brokers decouple senders and receivers.
- Since message brokers require the communication to go through a middle layer, even more application-specific functionality can be implemented there, e.g., *content transformation* rules.

The Publish/Subscribe Interaction Model

sebis

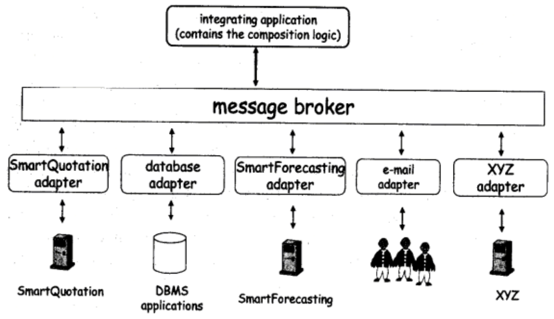
- Applications simply **publish** the message to the middleware system → **publishers**
- If an application is interested in receiving messages of a given type, then it must **subscribe** with the publish/subscribe middleware.
- Whenever a publisher sends a message of a given type, the middleware retrieves the list of all applications that subscribed to messages of that type, and delivers a copy of the message to each of them.



EAI with a Message Broker

sebis

- Adapters map heterogeneous data formats, interfaces, and protocols into a common model and format.
- A different adapter is needed for each type of application that needs to be integrated.



sebis

- “Conventional” Middleware for Distributed Information Systems [Al05]
  - RPC and Related Middleware
  - Object Brokers
  - Message-Oriented Middleware
  - EAI Middleware: Message Brokers
- Web Services
- Service-Oriented Architecture
- REST [Fi00]

Limitations of Conventional Middleware in B2B Integration (1)

sebis

- In cross-organizational interactions there is **no obvious place** where to put the middleware.
- The basic idea for conventional middleware was for it to reside **between the applications** to be integrated and to mediate their interactions.
- The applications were distributed, but the middleware was **centralized** (at least logically), and it was controlled by a **single company**.

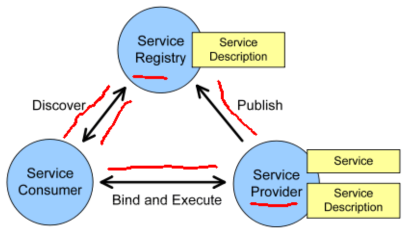
Roles and Actions in an SOA (1)

sebis

- A service oriented architecture has three involved roles: service **consumer**, service **provider**, and service **registry**.
- The service consumer
  - is an application, a software module or another service that requires a service,
  - initiates the enquiry of the service in the registry, binds to the service over a transport, and executes the service function.
- The service provider
  - is a network-addressable entity that accepts and executes requests from service consumers,
  - publishes its services and interface contract to the service registry so that the service consumer can discover and access the service.
- A service registry
  - is the enabler for service discovery,
  - contains a repository of available services and allows for the lookup of service provider interfaces to interested service consumers.

Roles and Actions in an SOA (2)

sebis



[Pa03]

Principles of a Service-Oriented Architecture [Er05]

- **Loose Coupling**: Services are using each others functionality while still remaining independent.
- **Service Contract**: A service in a SOA is described in a document which contains all information necessary for using it.
- **Discoverability** of services at design time enables their reuse.
- **Abstraction and autonomy**: A service hides implementation details and will be accessed only through its interface.
- **Reusability**: Services can be consumed by more than one consumers.
- **Composability**: A coarse grained service may orchestrate several service, which are of a finer granularity.
- **Stateless** services should minimize the amount of state information they manage, as well as the duration for which they remain stateful.

Services

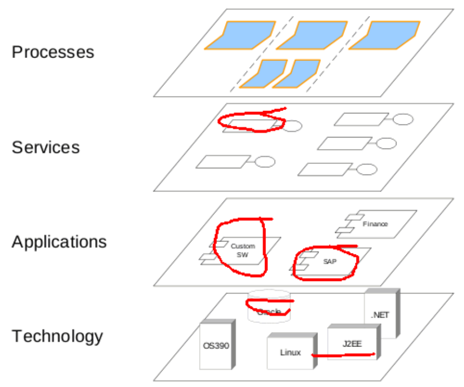
- A service provides some functionality over a standardized interface, which encapsulated the actual implementation.
- As with SOA, there are many different definitions of a service.

- A service can be divided into these parts [KBS04]:
  - Service description,
  - Service interface,
  - Business logic,
  - Implementation, and
  - Data.

SOA Governance

- Governance plays an important role in adopting and managing an SOA.
- SOA governance is important at three different levels [Kel07]:
  - At the **strategic** level the management of a company defines which role SOA should play.
  - At the **operational** level decisions, which may cross departments, but do not have influence on the whole organization are made:
    - Who is the owner of a service?
    - Who pays for implementation and maintenance of a service?
    - Which non-functional requirements have to be fulfilled by a service, e.g., availability, performance?
  - At the **technical** level tools help to ensure technical integrity.

Alignment of Business and IT



This is just a teaser: More information ➔ SEBA Master & Strategisches IT-Management

- "Conventional" Middleware for Distributed Information Systems [Al05]
  - RPC and Related Middleware
  - Object Brokers
  - Message-Oriented Middleware
  - EAI Middleware: Message Brokers
- Web Services
- Service-Oriented Architecture
- REST [Fi00]

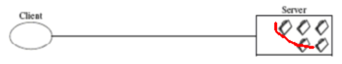
Representational State Transfer

- Representational State Transfer (**REST**) is a software architectural style for distributed hypermedia systems like the world wide web.
- The term has been coined by Roy Fielding in his doctoral dissertation [Fie00]
- REST provides a set of architectural constraints that, when applied as a whole, emphasizes
  - scalability of component interactions,
  - generality of interfaces,
  - independent deployment of components, and
  - intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.
- The REST architectural style has been used to guide the design and development of the architecture for the modern Web.
- The abstract discussion about architectural styles enables judgments over whether particular practices are consistent with the architecture of the Web.

The Null Style

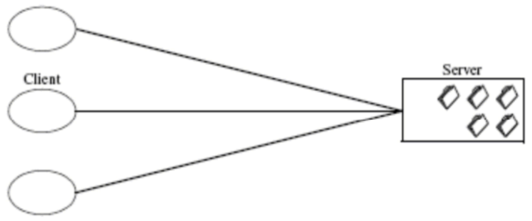
- Is simply an empty set of constraints
- Describes a system in which there are no distinguished boundaries between components
- Is the starting point for the description of REST
- Examples: Mainframe application, Desktop application, "Closed" Distributed System (e.g. World of Warcraft)

Client-Server



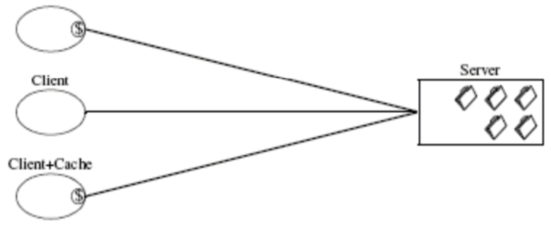
- Separates the user interface concerns from the data storage concerns.
- Improves the portability of the user interface across multiple platforms.
- Improves scalability by simplifying the server components.
- Allows the components to evolve independently.

Stateless



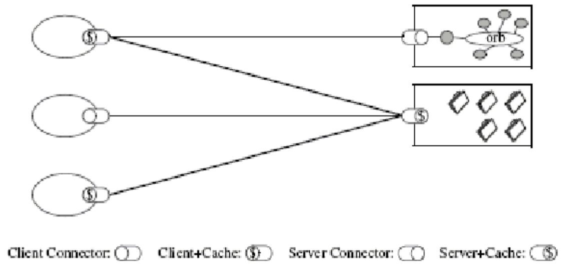
- Communication must be stateless in nature: each request from client to server must contain all of the information necessary to understand the request.
- Visibility is improved because a monitoring system does not have to look beyond a single request datum in order to determine the full nature of the request.
- Reliability is improved because it eases the task of recovering from partial failures.
- Scalability is improved because not having to store state between requests allows the server component to quickly free resources, and further simplifies implementation because the server doesn't have to manage resource usage across requests.
- Disadvantage: it may decrease network performance by increasing the repetitive data sent in a series of requests, since that data cannot be left on the server in a shared context.

Cacheable



- Require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable.
- Improves efficiency, scalability, and user-perceived performance by reducing the average latency of a series of interactions.
- Trade-off: a cache can decrease reliability if stale data within the cache differs significantly from the data that would have been obtained had the request been sent directly to the server.

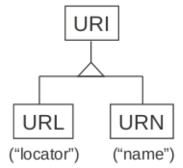
Uniform Interfaces



- This is the central feature that distinguishes the REST architectural style from other network-based styles.
- Emphasis on a uniform interface between components:
  - Uniform identification scheme,
  - Uniform representation of information exchanged.
- Implementations are decoupled from the services they provide.
- Trade-off: information is transferred in a standardized form rather than one which is specific to an application's needs.

Uniform Identification Scheme

- URI: Uniform Resource Identifier (general term)
  - There are two mechanisms: naming and location
- URN: Uniform Resource Name
  - Identification of objects by name for the purpose of persistent labeling
  - ~~E.g., ISBN~~
- URL: Uniform Resource Locator
  - Identification via the primary access mechanism



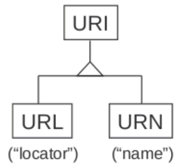
- An example of an URI:

scheme		authority	path		query		fragment
`http`	`://`	`example.org`	`/mysite/page`	`?`	`name=cat`	`#`	`whiskers`

- A URI points to a hierarchical space reading from left to right; each block that follows is a branch from the previous block.

Uniform Identification Scheme

- URI: Uniform Resource Identifier (general term)
  - There are two mechanisms: naming and location
- URN: Uniform Resource Name
  - Identification of objects by name for the purpose of persistent labeling
  - E.g., ISBN
- URL: Uniform Resource Locator
  - Identification via the primary access mechanism



- An example of an URI:

scheme		authority	path		query		fragment
`http`	`://`	`example.org`	`/mysite/page`	`?`	`name=cat`	`#`	`whiskers`

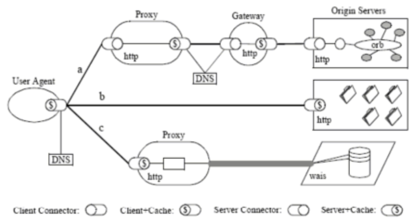
- A URI points to a hierarchical space reading from left to right; each block that follows is a branch from the previous block.

The URI Dissected

- **Scheme** – Defines how the URI should be interpreted.
- **Authority** – Has the structure userinfo@host:port
- **Path** – Looks like the path on a file system and is often used in a hierarchical fashion to address files on a system.
  - Example: http://del.iciou.us/danja/owl refers to all items tagged by danja with owl.
- **Query** – The spec describes the query part as being a non-hierarchical part of the URI.
- **Fragment** – Is used to identify a secondary resource.

- URIs are not being used uniformly on the Web, but: when a URI has been used to identify a resource, it should continue to be used to identify the same resource.
- See: "**Cool URIs don't change**" from Tim Berners Lee. [http://www.w3.org/Provider/Style/URI]

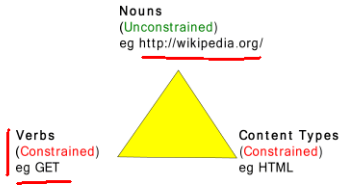
Process View of a REST-Based Architecture



- Request (a) has been sent to a local proxy, which in turn accesses a caching gateway found by DNS lookup, which forwards the request on to be satisfied by an origin server whose internal resources are defined by an encapsulated object request broker architecture.
- Request (b) is sent directly to an origin server, which is able to satisfy the request from its own cache.
- Request (c) is sent to a proxy that is capable of directly accessing WAIS, an information service that is separate from the Web architecture, and translating the WAIS response into a format recognized by the generic connector interface.
- Each component is only aware of the interaction with their own client or server connectors; the overall process topology is an artifact of our view.

REST Triangle

- The main problem domains identified in REST are the **nouns**, the **verbs**, and the **content-type** spaces.
- The *things that exist*, the *things you can do to them*, and the *information you can transfer* as part of any particular operation.



- REST requires a **standardized** set of state transfer operations.

REST Methods

- Minimum methods: GET, PUT, POST, DELETE
- GET is the HTTP equivalent of COPY
  - Transfers a representation from resource to client.
- PUT is the HTTP equivalent of PASTE OVER
  - Transfers state from a client to a resource.
  - GET and PUT are fine for transferring state of existing resources.
- POST is the PASTE AFTER verb
  - Don't overwrite what you currently have: Add to it
  - Create a resource.
  - Add to a resource.
- DELETE is the HTTP equivalent of CUT
  - Requests the resource state being destroyed.

REST versus RPC

```
getUser()
addUser()
removeUser()
updateUser()
getLocation()
addLocation()
removeLocation()
updateLocation()
listUsers()
listLocations()
findLocation()
findUser()
```

```
http://example.com/users/
http://example.com/users/{user} (one for each
user)
http://example.com/findUserForm
http://example.com/locations/
http://example.com/locations/{location} (one for
each location)
http://example.com/findLocationForm
```

```
userResource =
  new Resource("http://example.com/users/001")
userResource.get()
```

```
exampleAppObject =
  new ExampleApp("example.com:1234")
exampleAppObject.getUser()
```

- There are many examples of interfaces that label themselves 'REST', but are, in fact, using HTTP to tunnel function calls.

REST Methods

- Minimum methods: GET, PUT, POST, DELETE
- GET is the HTTP equivalent of COPY
  - Transfers a representation from resource to client.
- PUT is the HTTP equivalent of PASTE OVER
  - Transfers state from a client to a resource.
  - GET and PUT are fine for transferring state of existing resources.
- POST is the PASTE AFTER verb
  - Don't overwrite what you currently have: Add to it
  - Create a resource.
  - Add to a resource.
- DELETE is the HTTP equivalent of CUT
  - Requests the resource state being destroyed.

REST versus RPC

```
getUser()
addUser()
removeUser()
updateUser()
getLocation()
addLocation()
removeLocation()
updateLocation()
listUsers()
listLocations()
findLocation()
findUser()
```

```
http://example.com/users/
http://example.com/users/{user} (one for each
user)
http://example.com/findUserForm
http://example.com/locations/
http://example.com/locations/{location} (one for
each location)
http://example.com/findLocationForm
```

```
userResource =
  new Resource("http://example.com/users/001")
userResource.get()
```

```
exampleAppObject =
  new ExampleApp("example.com:1234")
exampleAppObject.getUser()
```

- There are many examples of interfaces that label themselves 'REST', but are, in fact, using HTTP to tunnel function calls.

End of presentation. Click to exit.