Architecture of Distributed Systems

Architectural Styles

Original: J.J. Lukkien
Revision: R.H. Mak
Goal of the lecture

**Students should understand:**
- What architectural styles/patterns are
- What their role in architectural design is

**Students should know:**
- For a number of specific styles:
  - The motivation for their usage
  - Their typical AEs, relationships
  - Rules and constraints for interaction
    - amongst AEs and with the environment
  - Weaknesses and strengths
- Some common concepts for distributed systems
  - Service, protocol, binding
Material

Mandatory
Van Steen, Tanenbaum, Distributed Systems, 3rd ed, Chapter 2

Background:
• Bass, Clements, Kazman, Software Architecture in Practice, 3rd ed., Chapter 13
• Rozanski and Woods, Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, 2nd ed, Chapter 11,
Architectural styles (patterns)

- An architecture is "the set of fundamental concepts or properties of a system in its environment" embodied by
  - its architectural elements (a.k.a. building blocks, components),
  - its relationships – amongst elements and between elements and the environment (connectors and interfaces, dependencies)
  - and the principles guiding its design (rules & constraints for building blocks and connectors and their composition) and evolution

- An architectural style is a coherent set of design decisions concerning the architecture
  - a combination of a typical (de)composition and
  - typical choices for connectors, components and behavior
  .... a generic solution for a class of problems

- We also have interaction styles, concerning just the interaction between building blocks
  - the nature of the connectors, and their organization
  - taking care of synchronization and communication between system components
Architectural style characterization

• **Defined by**
  • *motivation (guidelines, context)* for the application of the style
    – which extra-functional properties are achieved, and how
    – which problem class is solved
  • *vocabulary*
    – names for components (building blocks) and connectors, and for other concepts
  • *rules (constraints, responsibilities)* for components and connectors
  • *generic structure and behavior*
    – interfaces of components, and correspondent connectors
    – data distribution, protocols, control flow, data flow

  ....documented as a *profile (e.g. UML)*

• **When applied, a style yields a partial architecture**
  • in fact: styles classify architectures

• **Within an architecture, several styles can be applied**
  • and also, several alternatives in interaction style

• **Styles encourage communication, reuse, comparison (of alternatives)**
An architectural style a.k.a. architectural pattern.

Other software patterns are design patterns (GoF) and (language) idioms

A design pattern provides a scheme for refining the elements of a software system or the relationships between them. It describes a commonly recurring structure of interconnected design elements that solves a general design problem within a particular context.

A language idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of elements or relationship between them by using features of a given language.
Anti-patterns.

Coined by Andrew Koenig

An anti-pattern is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive

• includes applying an, in itself, useful pattern under the wrong circumstances or in an unusual way
• In addition there are simply bad practices that commonly appear but should always be avoided (bad smells)
Tactics versus Patterns

Tactics
• target individual quality attributes
  • but may influence others than the one they target as well

Patterns
• target common/re-occurring architectural problems
• address a range of quality attributes
  • often striking a trade-off
• comprise a set of tactics that work together towards solving a problem
Architectural styles

- **Data flow styles**
  - Batch sequential
  - Pipes & Filters

- **Layered styles**
  - Virtual machine
  - Interpreter

- **Independent components**
  - Client-Server
  - Peer to Peer
  - RESTful / ROA
  - SOA

- **Event-based**
  - Publish & subscribe
    - pure or brokered
  - Whiteboard

- **Data-centered**
  - Blackboard,
  - Databases (SQL, NoSql)
  - Model-View-Controller
  - Rule-based

- **Other**
  - Mobile agents
  - Tuple spaces

Classification varies widely amongst various authors
Interaction styles / communication models

- **Basic inter-process communication (IPC)**
  - message passing, streaming, sockets,
- **Remote invocation**
  - request –reply
  - procedure/method call
  - remote procedure call (RPC)
  - active messages
- **Indirect communication**
  - shared memory
  - eventing
  - message queues
- **Streams and files**
- **Interaction styles induce some architectural elements, e.g.,**
  - RPC infra structure, client / server libraries, middleware stacks
Viewpoints drive choice of styles

- While describing an architecture, various viewpoints are taken:
  - user interaction viewpoint
  - data viewpoint:
    - focus on how data flows through the system, and is stored
  - control viewpoint:
    - focus on control flow
  - decomposition viewpoint

- The relative importance of these viewpoints often determine choices of architecture styles:
  - i.e., architecture styles are best understood with respect to a viewpoint
Layered style

- **Motivation:**
  - independent layers of abstraction
  - separate concerns,
  - limit dependencies,
  - allow replacement

- **Vocabulary:**
  - layer module, service, interface (building blocks)
  - call, call-back, handle (interaction), allowed-to-use relation

- **Rules:**
  - Layer $N$ may know only layer $N-1$
    - not vice versa (why?)
    - layer $N$ is in control of binding the layer $N/N-1$ interface
  - Layer $N$ may not use knowledge of lower layers than $N-1$ (*strict layering!*)
  - Control flow can be both synchronous and asynchronous (“event”, “call-back”)

- **Structure:**

- **Typical behavior:**
  - (Objects/modules in) layer $N$ call upon the API of (objects in) layer $N-1$
    - regard layer $N-1$ as the single service provider of all lower functionality
  - Layer $N$ registers call-back routines with layer $N-1$
Client-Server style

• Motivation:
  – sharing some localized resource (e.g. file store, compute server)
  – protecting and managing content (e.g. a database)
  – delay binding, decrease dependencies (independent development)
  – generally: separation of concerns

• Vocabulary:
  – client, server (building blocks)
  – request, reply (interaction)
  – server discovery

• Rules:
  – Servers are passive
    • provides a service upon request from clients, does not know clients
    • handles data access, data integrity
  – Clients are active
    • initiate activity, discover servers
  – no connection among clients
  – no, or limited, state on the server per client

• Structure:

• Typical behavior:
  – client finds server access point through discovery
  – regular interaction:
    – possibly as part of a session
Application layers and C-S functionality

- 3-layer logical organization of data access
- cuts possible at many places: define tiers (physical layers)
  - (a,b) thin client – changing control
    - the user interface has small overhead; it typically runs on a machine that is a client to the application server
    - however, after connection, the user interface becomes a server for the application
  - (c,d) fat client
  - (e) file server
Combining layering and C/S in a distributed fashion

Development (logical): layering ("presentation, business and data tier (layer)")
Deployment: Client-Server interactions between the components

- also called: multi-tier client-server organization
- note: middle layer plays both client and server roles
  - that is actually what layering + request/reply is
Peer-to-peer style

• Motivation:
  – sharing resources and content
  – cooperation in communities
  – symmetry in roles
  – increase concurrency
  – horizontal scalability

• Vocabulary:
  – peer, super peer, distributed hash table (building blocks)
  – community (of peers)
  – overlay, application level multicast, peer discovery

• Rules:
  – Peer:
    • symmetric in functionality, and contribution
    • can discover, reach and cooperate with all other peers, in principle
    • can be both passive and active
  – Super peer
    • contributes extra resources to the community (extra services, like directory, discovery)

• Structure::
  – overlay, on top of network technology (from deployment view)

• Typical behavior:
  – peer finds community access point
  – joins community
  – provides passively services to other peers
  – actively uses services from peers in the community
  – the collective services of peers result in new functionality
    • e.g. file sharing

• Example: Skype VOIP
Combine peer-to-peer with application layering

- Vertical distribution: map entire *layers* to machines
- Horizontal distribution:
  - divide a *layer* across a collection of symmetric *peers* (picture)
  - e.g. a distributed database, distribution of business logic
  - divide different functions inside a layer across (asymmetric) peers
**Batch sequential style**

- **Motivation:**
  - Processing in several stages
  - Monolithic component would be too complex
  - Reconfigurable system: stages inserted/removed/reordered easily

- **Vocabulary:**
  - Processing steps (stages), batch of data (single input set), sequential processing

- **Rules:**
  - Every stage is stand alone in processing. There is no shared state.
  - Only subsequent stages exchange data.
  - System handles input sets one by one. No stream of data. No concurrent work of stages on different data.

- **Weak point**
  - Not interactive and slow due to sequential processing

- **Structure:**
  - Data Input
  - Stage 1 → Stage 2 → Result

- **Typical behavior:**
  - Stages designed to be reusable (e.g. have configurable parameters)
  - Different stages can be on different processors, but their execution is still sequential.

- **Metaphore**
  - Assembly line that can start processing next product only after previous one has been completely processed

**Examples:** Compilers, classical ETL systems
Pipes & filters style

• Motivation:
  – Systems that work with streams of data (e.g. video processing multimedia systems)
  – Streams of data are naturally processed in several stages
  – Reconfigurable system: stages inserted/removed/reordered easily

• Vocabulary:
  – Pipes, filters, data stream, data source, data sink, ports, buffers, pipeline
  – Data stream is processed by filters
  – Filters are connected via pipes (explicit connectors).
  – Forks and joins are allowed, but pipeline is sequence of filters from data source to sink.

• Rules:
  – Every filter is stand alone stage in processing. There is no shared state.
  – Pipes store data (state) and preserve or ordering (FIFO)
  – Only adjacent filters exchange data

• Weak points:
  – Difficult to share global data
  – Difficult to handle control (e.g. filter crashes)
  – Not convenient for interaction

• Structure:

• Typical behavior:
  – Filters are designed to be reusable (e.g. have configurable parameters)
  – Filters can execute concurrently (on different processors) and asynchronously
  – Pipes have finite capacity; filters block on read and possibly on writes (otherwise data loss). Capacity usually chosen to prevent this

• Metaphore:
  – Stream of products being processed along assembly line
  – Example: Unix shell commands, GStreamer
Publish/subscribe style

• Motivation:
  – Decoupling data producer from data consumer
  – Sending data when it is available. Avoiding need to poll for data.
  – Allow multiple producers/consumers
  – Allow runtime changes of set of producers/consumers

• Vocabulary:
  – Publisher, subscriber, subscription, notification, topics, broker
  – Subscriber registers with publisher to receive notifications for chosen topic.

• Rules:
  – Every subscription relates a topic to 1 subscriber.
  – Publisher can have multiple subscribers for same topic.
  – Notification goes to all subscribers; however, subscribers may specify receive policies
  – When existent, broker decouples the publishers and subscribers

• Structure::

• Typical behavior:
  – Subscribers find brokers and send subscriptions for topics of interest
  – Publisher register with brokers, or are discovered
  – Notification may be only about events, or it can also contain data.
  – Broker can be omitted

• Metaphore:
  – Newspaper/magazine subscription
  – Observer design pattern is an example of publish/subscribe style
Model – View - Controller style

- **Motivation:**
  - Multiple views in interactive applications need to be kept synchronized
  - ‘Look and feel’ of user views tends to change a lot over time (sometimes even underlying technology is changed)
  - Decoupling views from model facilitates change of views without need to rewrite model as well

- **Vocabulary:**
  - User interacts with system via views
  - Model contains all data represented by views.
  - Controller updates the model
  - Views present model to user

- **Rules:**
  - View does not handle user input. It forwards it to controller
  - Only controller can initiate updates of the model
  - Model does not change views. It notifies them about change
  - Views query relevant parts of model and update their representations of it

- **Structure::**

- **Typical behavior:**
  - User interacts with a view resulting in model changes through controller
    - Possible: separate controller for every view
  - All views are updated accordingly
    - Notification mechanism can be based on publish-subscribe style

- **Weak points:**
  - Changing model has consequences for most or all views
  - Not convenient for very frequent events
  - Too complex for simple interfaces
Blackboard style

• Motivation:
  – different tasks on same state (data set)
  – current state is result of (serialized) operations of different tasks
  – tasks are coupled only through the shared state; communicating partners do not know each other

• Vocabulary:
  – blackboard, knowledge sources, control component, data space
  – (‘knowledge-source’ derived from the envisaged domain of problem solving)

• Rules:
  – Blackboard (data space):
    • contains all state
    • is passive
  – Control
    • transfers control among KS’s
    • conflict resolution
  – Knowledge source
    • algorithms for state adjustment
    • generation of new data
    • don’t know each other

• Structure::
  – knowledge sources access blackboard under control of Control

• Typical behavior:
  – KS’s register with Control
  – Repeatedly, Control determines enabled KS’s, selects and calls one to execute

• Metaphore
  – several people solving a problem jointly, by writing on a blackboard
Shared Data Space: Data Distribution Service (DDS)

- **OMG (Open Management Group) standard**
  - [http://portals.omg.org/dds/what-is-dds-3/](http://portals.omg.org/dds/what-is-dds-3/)
- Data-Centric Publish-Subscribe (DCPS) – low level interface
  provides a global data space. Type-specific data publishers send data that interested subscribers can receive.
- Data Local Reconstruction Layer (DLRL) - optional layer built on top of DCPS
  DLRL providing more transparency by hiding much of the pub/sub details and allowing for distribution of an object model.

- **Implementations**
  - open source: OpenDDS (implements DCPS), OpenSplice (real-time variant)…
Service oriented (SOA) style

**Motivation:**
- Separate *functionality* ("service"), the *implementation*, the *deployment context* and the *application context*
- Build applications by very late (dynamic) binding
- Integration of enterprise information systems

**Vocabulary:**
- SOA, service, interface, discovery, registry, composition, binding, orchestration, choreography, SLA

**Rules:**
- Application is built ad-hoc out of services that communicate in a standardized manner
  - via a network
  - see e.g. REST
- Service is a self-contained functionality. It does not depend on state of other services, or of the platform (OS, language) it is running on.
- Services are discoverable

**Structure: (conceptual):**

**Typical behavior:**
- Providers publish services; Applications ("orchestrations") discover services, and bind their interfaces
  - Service broker (registry) can exist to manage discovery process
- Applications send data objects through a number of services as a workflow
  - Often XML based RPC (SOAP)
  - Services are typically kept simple and focused on single task

**SOA is a huge research domain in BIS**
Enterprise application organization

Monolithic applications

Tiered service architecture
Enterprise application organization

Monolithic applications

Tiered service architecture
Example: UPnP (part of DLNA)

- **Universal Plug’n Play**
- **Two types of entities:** *devices* and *control points*.
  - Devices are service providers, control points are service users.
- **Discovery protocol:**
  - A device added to the network can advertise its services.
  - A control point added to the network can search for devices (and then services) of interest.
- **Description:**
  - A control point can obtain a detailed (XML) description of a device. Descriptions contain information about services offered by a device and their interface.
- **Control points**
  - can invoke actions offered by services
  - can subscribe to events published by services
  - and can access URLs with presentation provided by devices
  - can bind interfaces of services
Logical organization, interaction

A typical UPnP application uses an API to use functions of Remote Procedure Call and service discovery. Standard protocols are used for the core protocols of the standard.

Interactions: control points call actions on services (top) or establish connections between services (bottom).
REST: REpresentational State Transfer

- **Motivation:**
  - special usage of C&S aiming at
    - portability
    - independent development & deployment
    - reduction of interaction complexity
    - reliability
    - scalability

- **Vocabulary:**
  - user agent, origin server, gateway, proxy
  - cache, layer
  - state
  - code-on-demand
  - resource, resource identifier (e.g. URL), representation, metadata

- **Rules:**
  - stateless communication (no server state)
  - response is labeled as cache-able, and can then be cached
  - uniform interface between components
    - decouples structure from functionality
    - layering
    - client functionality can be adapted by code-on-demand

- **Structure:**

- **Typical behavior:**
  - as (layered) C&S, but only a single service (interface)

- **Prime example: WWW**
Virtual machine style

**Motivation:**
- Portability (eliminate platform dependencies)
- Compact task representation
- Code mobility

**Vocabulary:**
- Platform, virtual machine (VM), native code, byte code, compiler, memory, state of VM, state of program
- Programs are written in native code. This human understandable form is compiled to byte code, interpreted by a Virtual machine.

**Rules:**
- Actual execution of program(s) is inside virtual machine
- Same byte code can run on any platform (OS, ISA). Virtual machine needs to be ported.

**Metaphor:**
- A speaker needs to hold his speech in different countries. He translates his speech into English. Speaker reads English version to audience, while translator converts this to the language of the audience.

**Structure:**

**Typical use**
- Compile once (locally), run anywhere
  - part of REST (mobile code)
- virtual machine provides API for access to the local platform

**Weaknesses:**
- Performance penalty (e.g., on same machine Java programs run typically slower than C/C++ programs)
Interpreter style

**Motivation:**
- (Same as for VM)
- Domain-specific language, with high-level concepts from the domain

**Vocabulary:**
- Platform, interpreter (execution engine), interpreted program, pseudo-code (script), memory, state of engine, state of program
- Programs are written in a high level pseudo code or script. Execution engine interprets this code.
- Main difference with VM: absence of intermediate machine language

**Rules:**
- Actual execution of program(s) is inside interpreter
- Same pseudo code can run on any platform. Execution engine needs to be ported.

**Weaknesses**
- Performance penalty
  - Larger than for Virtual machine as there is no machine language that can be optimized

**Structure:**

**Typical behavior:**
- Programmer writes script with high expressive power (awk, perl...)
- Scripts are deployed to a variety of machines

**Metaphore**
- A speaker needs to hold his speech in different countries. Speaker talks in his own language to audience, while translator simultaneously converts this to the language of the audience.
Choosing an architectural style

• No single solution: the same problem can be tackled with many different designs and architectural styles

• Choice depends often on extra-functional requirements:
  • Costs (resource usage, development effort needed)
  • Scalability (effects of scaling work complexity and available resources)
  • Performance (e.g. execution time, response time, latency..)
  • Reliability
  • Fault tolerance
  • Maintainability (extending system with new components)
  • Usability (ease of configuration and usage…)
  • Reusability
Architecture style application

- An *(architectural) framework* supports implementing instances of a style
  - it supports the concepts, the vocabulary of that style
  - it defines (part of) the development *procedure*, it *automates* the *generation* of standard *components* and *interfaces* as well as the *application* of the *constraints* (rules)
  - it supports the interaction styles deemed suitable for the style and automates their use
    - we can also have a framework just for the interaction style, e.g. an RPC framework

- The concept of a framework is defined in the ISI/IEC/IEEE 42010 standard and may include an ADL and tool support.

- A framework defines part of the *life cycle* of the software or system it is used for

- A framework may include a programming model
Life cycle

- The *life cycle* of a product or system is the series of stages it goes through from *inception* to *decommissioning*.

- A typical life cycle for a software system is given to the right.

- More detail is obtained by adding information regarding the activities in the stages.

- Notes:
  - also system parts have life cycles, affecting the overall system
  - the life cycle should also address evolution, redesign
Programming model

• A Programming Model is set of concepts, primitives and combinators to express a computation or system behavior
  • often by means of an API or as a library

• Examples:
  • the language C: variables, functions, assignments, expressions, function calls, repetition, selection, sequential composition
  • C++ adds objects, classes, inheritance as concepts, and primitives to use them
  • an RPC programming model has concepts and primitives to manage and express an RPC-based system
  • Google’s MapReduce: functions Map and Reduce
  • Spark Lineage graphs
  • parallel and functional programming models
    – CSP, MPI, Unity, Dataflow; Haskell, Perl, Prolog
  • database query languages such as SQL
<table>
<thead>
<tr>
<th>Framework</th>
<th>Domain</th>
<th>(Dominant) Pattern</th>
<th>Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ruby on Rails</td>
<td>Web applications</td>
<td>MVC</td>
<td>Ruby</td>
</tr>
<tr>
<td>Spring</td>
<td>Enterprise applications</td>
<td>MVC</td>
<td>Java</td>
</tr>
<tr>
<td>Hadoop</td>
<td>Big Data Analytics</td>
<td>Master-slave</td>
<td>Map-Reduce</td>
</tr>
<tr>
<td>Gstreamer</td>
<td>Multimedia applications</td>
<td>Pipes &amp; filters</td>
<td>C + Glib</td>
</tr>
<tr>
<td>Avro</td>
<td>Large scale data processing and serialization</td>
<td>RPC</td>
<td>Json IDL</td>
</tr>
<tr>
<td>.Net</td>
<td>Web/Enterprise applications</td>
<td>SOA</td>
<td>Multiple language bindings: C#, C++, Jscript, Visual Basic, ...</td>
</tr>
</tbody>
</table>
Interfaces and services

• Apollo-Soyuz project (’75)
  • connect Apollo and Soyuz in space

• Needs clear definition of the interfaces
  • structure
    – ridges, rings, edges must fit
  • functionality (service)
    – the expected and provided responses of active elements
    – flow of control

• Adapter: extra component to reconcile two interfaces
  • translation, sometimes state holding

Smithsonian Air and Space museum, Washington D.C.
Definition: service

- **Service:** a contractually specified overall functionality (semantics) of an entity.

- **Service quality:** extra-functional properties of a service (e.g. speed, reliability, ...).
  - specified in a service level agreement (SLA)

- **Service interface (API):** actions (“primitives”) and responses that make the service available; these responses can be autonomous (“events”, “call-backs”). In addition, a specification that
  - describes their effect on state variables and parameters, as well as their results;
  - describes rules as how and in what sequence to call them;
  - describes the functional and non-functional properties of sequences of calls.
  *(i.e., the interaction or access protocol)*
The four methods of HTTP

- Not a REST standard (there are none), but in practice the only one interface used

For each resource the effect of the HTTP operation needs to be defined (see picture)

Get, Put and Delete are idempotent, Post is not

Use Post to

- Annotate existing resources
- Submitting a form to be processed
- Appending data to a resource
  - this may create a child resource
- In general, to modify a resource

Picture taken from: http://www.infoq.com/articles/rest-introduction
Definition: protocol

A protocol is a formal set of rules that dictates how information exchange, as well as interaction between entities (can be devices, execution threads, etc.), should take place.

• The rules specify
  • the format of the messages exchanged;
  • a finite state machine, specifying the different protocol states and which messages are allowed to be sent in each state; these states determine, among others, the order of the messages.
  • timing constraints and other quality properties, if any.

• Note:
  • Here ‘a message’ includes parameters passed upon function calling and event notifications
  • one may specify a protocol without being explicit about the overall service realized by it
Service example: file i/o

- **Service**: the complete functionality of being able to use files
  - store and retrieve data
  - create, destroy

- **Interface**: the calls that make this available
  - open, read, write, close, ...

- **Quality**: e.g. speed, level of control (buffers), protection

- **Rules**: calling sequences (open/create, read, ...), argument meanings, dealing with ownership, interpretation of return values
  - the access protocol
Service example: The X-Window System

- Application servers are **clients** to the X-kernel, which is a **server** for advanced window operations.

- **Xlib**: library that offers as a service:
  - the concepts of screen, windows, pixels, keyboard and mouse events
  - operations for manipulation of these
  - abstraction level is variable (pixels .. window)

- **The X protocol** is application independent

- The manager of the screen (to manage the windows) is just another client

- Application servers can run anywhere
  - locally, or at the X server side
  - enables ‘thin client’ organization: just user interaction
Example: UPnP services

- Originally UPnP services are accessed using SOAP but also a REST-like style is possible
  - see, e.g., the paper by Newmarch

- Service implementation is entirely hidden, as are the OS and the implementation language

Interactions: control points call actions on services (top) or establish connections between services (bottom)
The functionality and properties that a protocol provides is called the *provided service*. The provided service is often specified by an API for the protocol.

A protocol can be realized directly using basic services (e.g. hardware transport, machine instructions). If not, it relies on another service and corresponding protocol, called the *carrier*.

A protocol requires services from its carrier and any carrier providing these services can be used. The set of rules that specify how a protocol is mapped onto a carrier is called a *binding*.

In general, *binding* refers to establishing a relation between a reference and a referred object.
  - this interpretation of the term is slightly different from its meaning in “protocol binding”
  - usually, the correct meaning can be derived from the context
Example: layering in communication

- Layer 1: Physical medium
  - Layer 1 protocol
  - Layer 1 interface binding

- Layer 2:
  - Layer 2 protocol
  - Layer 2 interface binding
  - Provided service

- Layer 3:
  - Layer 3 protocol
  - Required service

- Layer 4:
  - Layer 4 protocol

- Layer 5:
  - Layer 5 protocol

Layers 1/2 to 4/5 interface
Distinguish service from protocol
Example: SOAP + XML over HTTP

Carrier protocol
Alternative SMTP

POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice xmlns:m="http://www.example.org/stock">
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
SOAP protocol states

Taken from:  [http://www.w3.org/TR/soap12-part2/]


TvS 2nd ed.: 2.1, 2.2.1, 2.2.2, 3.3, 12.1, 13.1, 13.2.1 (not exhaustive)

vST 3rd ed.: 2.1, 2.2 except inceptors, 2.3, 3.3 X-windows