Service Oriented Architecture & Web Services (part II)

Jan Jusko

FEE CTU

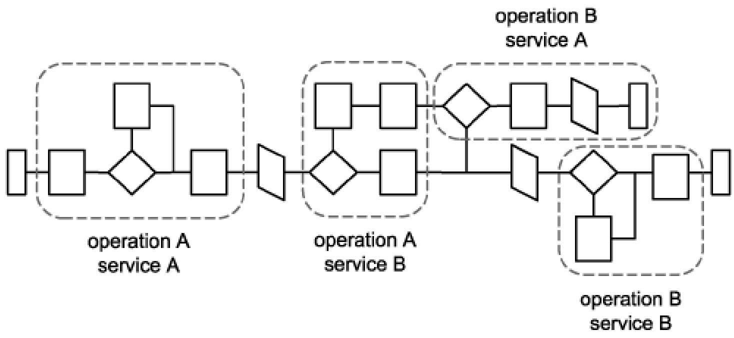
April 19, 2011

Outline

SOA building blocks

- **message**
  - unit of communication
  - represents the data required to complete some or all parts of a unit of work
- **operation**
  - unit of work
  - represents the logic required to process messages in order to complete a unit of work
- **service**
  - unit of processing logic (collections of units of work)
  - represents a logically grouped set of operations capable of performing related units of work
- **process**
  - unit of automation logic (coordinated aggregation of units of work)
  - represents a large piece of work that requires the completion of smaller units of work

SOA building blocks



WS counterparts

- SOAP messages
- Web service operations
- Web services
- activities
  - represent the temporary interaction of a group of Web services

SOA Principles

- reusability
- share a formal contract
- loosely coupled
- underlying logic abstraction
- composability
- autonomy
- discoverability
- statelessness

WS support for SOA principles (I)

- reusability
  - not automatically reusable, depends on the logic encapsulation
- **share a formal contract**
  - service descriptions (WSDL) are fundamental part of WS communication
- **loosely coupled**
  - naturally loosely coupled due to the use of service descriptions
- **underlying logic abstraction**
  - natively supported as Web Services publish only they interface and hide all the underlying logic

WS support for SOA principles (II)

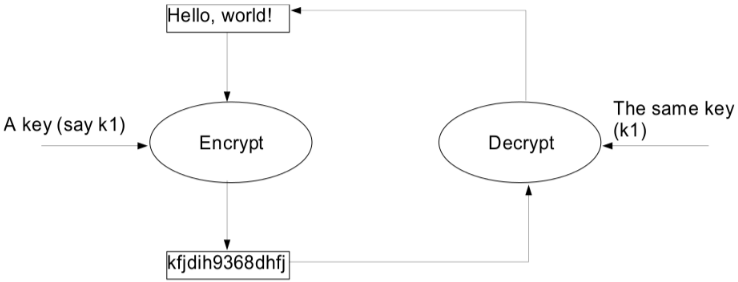
- **composability**
  - naturally composable, the extent of possible composability depends on the services design
- autonomy
  - requires design effort, not automatically autonomous
- discoverability
  - must be implemented by the architecture
- statelessness
  - preferred type of Web Services, but not guaranteed

SOA Principles not natively supported by WS

- reusability
- autonomy
- discoverability
- statelessness

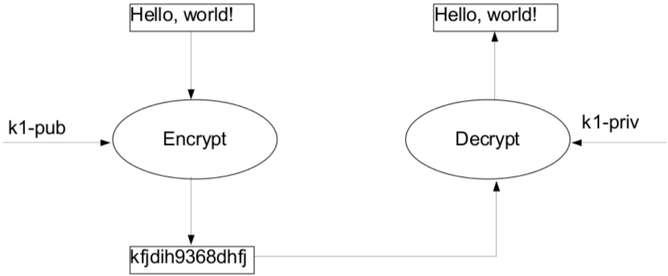
Symmetric encryption

- encryption and decryption use the same key



Asymmetric encryption

- uses a pair of keys, a *public* and a *private* key
- message encrypted by a public key can be decrypted only by a private key and vice-versa



Asymmetric encryption performance issue

- asymmetric cryptography is very computationally extensive
- use of a random encryption key
  - sender generates a random key
  - sender encrypts the generated key with recipient's public key and sends it
  - recipient decrypts the generated key
  - the generated key is then used to encrypt/decrypt the actual data to be sent

Hash Function

- a hash function H is a transformation that takes an input m and returns a fixed-size string, which is called the hash value h (that is, h $= H(m)$)
- the basic requirements for a cryptographic hash function are
  - the input can be of any length
  - the output has a fixed length
  - $H(x)$ is relatively easy to compute for any given x
  - $H(x)$ is one-way
  - $H(x)$ is collision-free
- used for example for checksums

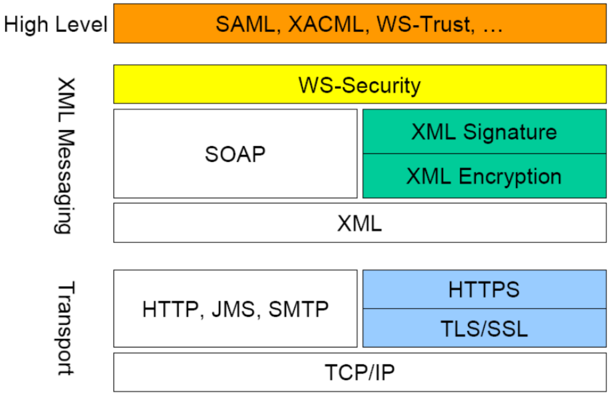
WS security - motivation

- **integrity** - messages are not duplicated, modified, reordered, etc.
- **confidentiality** - protects communication and data from passive attacks as eavesdropping or disclosure
- **authentication** - allows agents to prove their identity to each other, i.e. to verify that the opposite side of communications is who it claims to be

WS security

- transport layer security
- WS-security (XML/SOAP security)
- higher-layers security

SOA building blocks



WS security - transport layer

- well-known and established protocols
- point-to-point security
- request and response use same security properties
- transport specific

WS Security

- WS-security describes three main mechanisms
  - how to sign SOAP messages to assure integrity
  - how to encrypt SOAP messages to assure confidentiality
  - how to attach security tokens to ascertain the sender's identity
- uses cryptography, XML encryption and signatures

WS security - XML signature

- used to prove the identity of the sender & that the message is intact
- XML encryption is not an option as it is slow (calculation & transfer)
- instead, we calculate the hash of XML and encrypt it with our private key
- this encrypted hash is appended to the XML file and sent to the recipient
- the recipient decrypts the hash using the public key and compares with a hash value that he calculated
- if they are the same, the identity of the sender is verified as only the sender has the private key

WS security - XML encryption

- we can encrypt whole XML, a single element or contents of an element
- end to end security
- we can use symmetric or asymmetric encryption
- different security mechanisms can be applied to request and response
- self-protecting message (transport independent)

original XML

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://foo.org/details'>
  <Name>Joe User</Name>
  <CreditCard Limit='12,000' Currency='EUR'>
    <Number>1234 5678 9012 3456</Number>
    <Issuer>Local Bank</Issuer>
    <Expiration>12/06</Expiration>
  </CreditCard>
</PaymentInfo>
```

encrypted XML element and its contents

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://foo.org/details'>
  <Name>Joe User</Name>
  <EncryptedData Type='http://www.w3.org/...xmlenc#Element'
   xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <CipherData>
      <CipherValue>A23B45C56</CipherValue>
    </CipherData>
  </EncryptedData>
</PaymentInfo>
```

encrypted XML element contents

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://foo.org/details'>
  <Name>Joe User</Name>
  <CreditCard Limit='12,000' Currency='EUR'>
    <EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
     Type='http://www.w3.org/2001/04/xmlenc#Content'>
      <CipherData>
        <CipherValue>A23B45C56</CipherValue>
      </CipherData>
    </EncryptedData>
  </CreditCard>
</PaymentInfo>
```

encrypted XML

```
<?xml version='1.0' ?>
<EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
  Type='http://www.isi.edu/in-notes/...'>
    <CipherData>
      <CipherValue>GEwsRe234f</CipherValue>
    </CipherData>
</EncryptedData>
```

SOA delivery strategies

- top-down
- bottom-up
- agile
- **not to be mistaken with WS development strategies!**

SOA delivery strategies : top-down (I)

1. define ontology
   - identify concepts & entities and relationships among them
   - defines a new vocabulary that can be used to describe the problem domain
2. align business-models to the ontology
   - business-models might need to be adjusted to reflect new ontology
   - new business-models might be created
3. perform service-oriented analysis
4. perform service-oriented design
5. develop the services
6. test
7. deploy

SOA delivery strategies : top-down (II)

- analysis-first approach
- in general results in a high-quality service architecture
- very time-consuming and expensive
- might not show any immediate results

SOA delivery strategies : bottom-up (I)

1. model required application services
   - definition of application requirements that can be fulfilled through the use of WS, e.g. communication channel between legacy systems or B2B
2. design the required application services
   - limited space for design possibilities as the solutions may be purchased or automatically generated (wrappers)
   - new services should be modeled
3. develop the required application services
4. test
5. deploy

SOA delivery strategies : bottom-up (II)

- WS are built on as-needed basis
- WS are modeled to encapsulate application logic to best serve the immediate needs
- the most common approach
- SOA principles are rarely considered, not a true SOA

SOA delivery strategies : agile

- top-down and bottom-up approaches can be considered to be two extremes on the opposite sides of the spectrum
- seeking something in-between, that would incorporate proper SOA solution, while still providing quick delivery of services
- more complex than previous approaches
- business-level analysis concurrent with service design & development
- the process starts with business-analysis and after it has proceeded enough, the design phase starts as well
- developed processes need to be realigned after each cycle of business-analysis
- this approach requires much more effort, as developed services often need to be designed
- **immutable service contracts**
  - contract once published can not be changed, however, it can be extended

SOA analysis

- the process of determining how business automation requirements can be represented through service-orientation
- trying to answer the following questions:
  - what services need to be built?
  - what logic should be encapsulated by each service?
- goals of service-oriented analysis
  - define a preliminary set of service operation candidates
  - group service operation candidates into logical contexts. These contexts represent service candidates
  - define preliminary service boundaries so that they do not overlap with any existing or planned services.
  - identify encapsulated logic with reuse potential
  - define any known preliminary composition models

3 steps of SOA analysis

- define business automation requirements
- identify existing automation systems
  - any existing systems supporting the automation logic
  - legacy applications
  - this step helps identify application service candidates
- model candidate services
  - operation candidates are identified and grouped by logical context, thus creating services
  - services are further assembled into a composite model

General Design Pattern Template

- Problem
- Solution
- Application
- Impacts
- Principles

Design Pattern Groups

- Service Inventory Design Patterns
- Service Design Patterns
- Service Composition Design Patterns

Enterprise Inventory

- **Problem** Delivering services independently establishes a risk of producing inconsistent service and architecture implementations, compromising recomposition opportunities
- **Solution** Standardized, enterprise-wide inventory architecture wherein services can be freely and repeatedly recomposed.
- **Application** Modeled in advance, enterprise-wide standards are applied
- **Impacts** upfront analysis, organizational impacts
- **Principles** service contract, abstraction, composability

Domain Inventory

- **Problem** Enterprise directory is unmanageable
- **Solution** Grouping services into manageable domain-specific inventories, independent of each other
- **Application** Inventory domain boundaries need to be carefully established
- **Impacts** Standardization disparity between domain service inventories imposes transformation requirements and reduces the benefit of the SOA adoption
- **Principles** service contract, abstraction, composability

Service Normalization

- **Problem** When delivering services, there is a risk that services will be created with overlapping functionality, making reuse difficult
- **Solution** The service inventory needs to be designed with an emphasis on service boundary alignment
- **Application** Functional service boundaries are modeled as part of a formal analysis process
- **Impacts** Ensuring that service boundaries are and remain well-aligned introduces extra up-front analysis
- **Principles** service autonomy

Basics of Service Design Patterns

- most essential steps required to partition and organize solution logic into services and capabilities in support of subsequent composition
- *Service Identification Patterns* – The overall solution logic required to solve a given problem is first defined, and the parts of this logic suitable for service encapsulation are subsequently filtered out
- *Service Definition Patterns* – Base functional service contexts are defined and used to organize available service logic.

Functional Decomposition

- **Problem** To solve a large, complex business problem a corresponding amount of solution logic needs to be created -¿ self contained application
- **Solution** The large business problem can be broken down into a set of smaller, related problems
- **Application** Service oriented analysis is used to decompose the large problem
- **Impacts** The ownership of multiple smaller programs can result in increased design complexity

Service Encapsulation

- **Problem** Solution logic designed for a single application environment is typically limited in its potential to interoperate with other parts of an enterprise
- **Solution** Solution logic can be encapsulated by a service so that it is capable of functioning beyond the boundary for which it is initially delivered
- **Application** Solution logic suitable for service encapsulation needs to be identified
- **Impacts** Service-encapsulated solution logic is subject to additional design considerations

Service Façade

- **Problem** The coupling of the core service logic to contracts and implementation resources can inhibit its evolution
- **Solution** A service façade component is used to abstract a part of the service architecture
- **Application** A separate façade component is incorporated into the service design
- **Impacts** The addition of the façade component introduces design effort and performance overhead
- **Principles** service contract, service loose coupling

Redundant Implementation

- **Problem** A service that is being actively reused introduces a potential single point of failure
- **Solution** Reusable services can be deployed via redundant implementations
- **Application** The same service implementation is redundantly deployed or supported by infrastructure with redundancy features
- **Impacts** Extra effort is required to keep all redundant implementations in sync
- **Principles** service autonomy

References

- A4M33AOS materials by Jiri Vokrinek (http://cw.felk.cvut.cz/doku.php/courses/a4m33aos/start)
- Service-Oriented Architecture: Concepts, Technology, and Design by Thomas Erl
- Web Services Security by Mark O'Neill et al.