Service qualities

Service-Oriented Architecture

Jeremy Gibbons

Contents

1 Transactions

- idealized *indivisible activities*
- techniques for maintaining the illusion in the face of complexity, concurrency, failures
- ideas arose from distributed databases
- underlying finance, logistics, manufacturing...
- *Transaction Processing: Concepts and Techniques*, Gray and Reuter, 1993 (`http://books.google.co.uk/books?id=S_yHERPRZScC`)

1.1 ACID properties

- *atomicity*

  all-or-nothing

- *consistency*

  integrity-preserving: invariants satisfied

- *isolation*

  hidden intermediate results: multi-user behaviour consistent with single-user mode

- *durability*

  permanent committed results

1.2 Problems avoided

- *lost update*

  write committed and acknowledged but then discarded

- *inconsistent retrieval*

  reads of multiple fields at different times

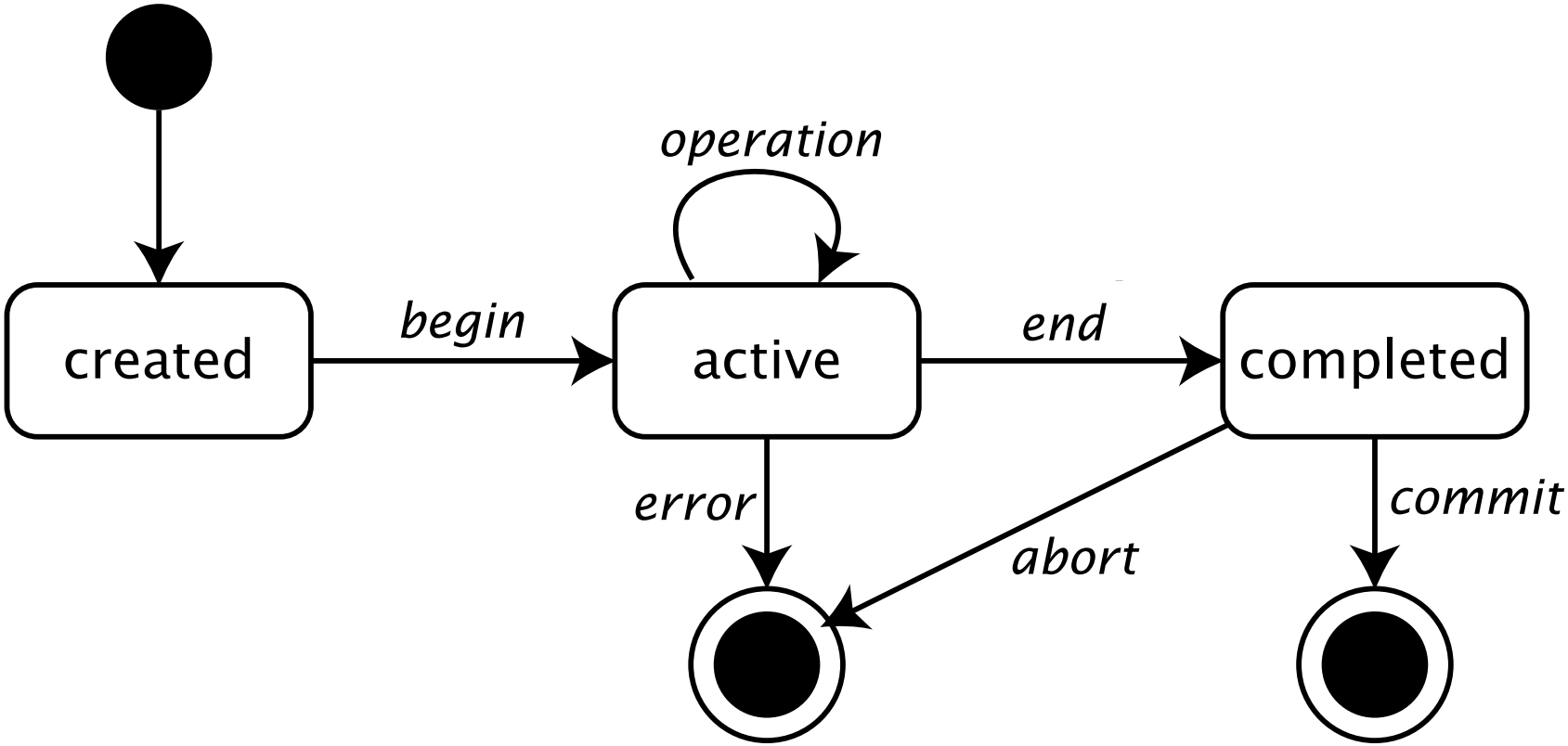
- *non-serializability*

  loss of single-user abstraction

- *conflict*

  eg simultaneous bookings of the same room

1.3 Transaction lifecycle



1.4 Two-phase commit

- *initiator* for each transaction, but any participant may abort
- 2PC protocol minimizes unavailability of unilateral abort
- *commit-request* phase

  initiator sends *commit?* message to all participants, who vote either *yes!* or *no!*
- *commit* phase

  initiator sends *commit!* (if unanimously *yes!*) or *abort!* (otherwise) to all participants, who act and *acknowledge*; then initiator completes transaction
- some disadvantages (blocking during 2PC; central control) so many variants...
- cf Christian wedding ceremony

1.5 Locking

- *serialization mechanisms* for resources, enforcing unique access
- *read locks* (shareable) and *write locks* (exclusive)
- may need to wait for locked resource to be released
- may result in *deadlock*: two parties, each waiting for the other
- lock resources in canonical order (requires foresight), or abort one party (requires rollback)
- chance of conflict rises as square of degree of multiplicity, and fourth power of size of transaction
- locks of varying granularity: arrange into DAG, lock from root to leaf and release in opposite order (coarser locks reduce overhead, but increase contention)
- see Gray, *Notes on DB OSes*, DOI 10.1007/3-540-08755-9_9

1.6 Nested transactions

- expensive transaction fails part-way through
- abort wastes much useful work
- how to avoid this, while staying acidic?
- organize transaction into tree of sub-transactions
- when sub-transaction commits, results visible only to parent
- if transaction rolls back, so do all its sub-transactions
- note: sub-transactions only ACI, not D

1.7 Long-running transactions

- lasting days, not seconds: *sagas*
- too expensive to holds locks, or they're not available (eg for human participants)
- optimism+compensation rather than pessimism+roll-back
- compensation usually only approximates 'undo'
  (cancellation fee, overstocking,...)
- misnomer: neither A nor I, and C must take non-I into account

1.8 WS-Transaction (WSTX)

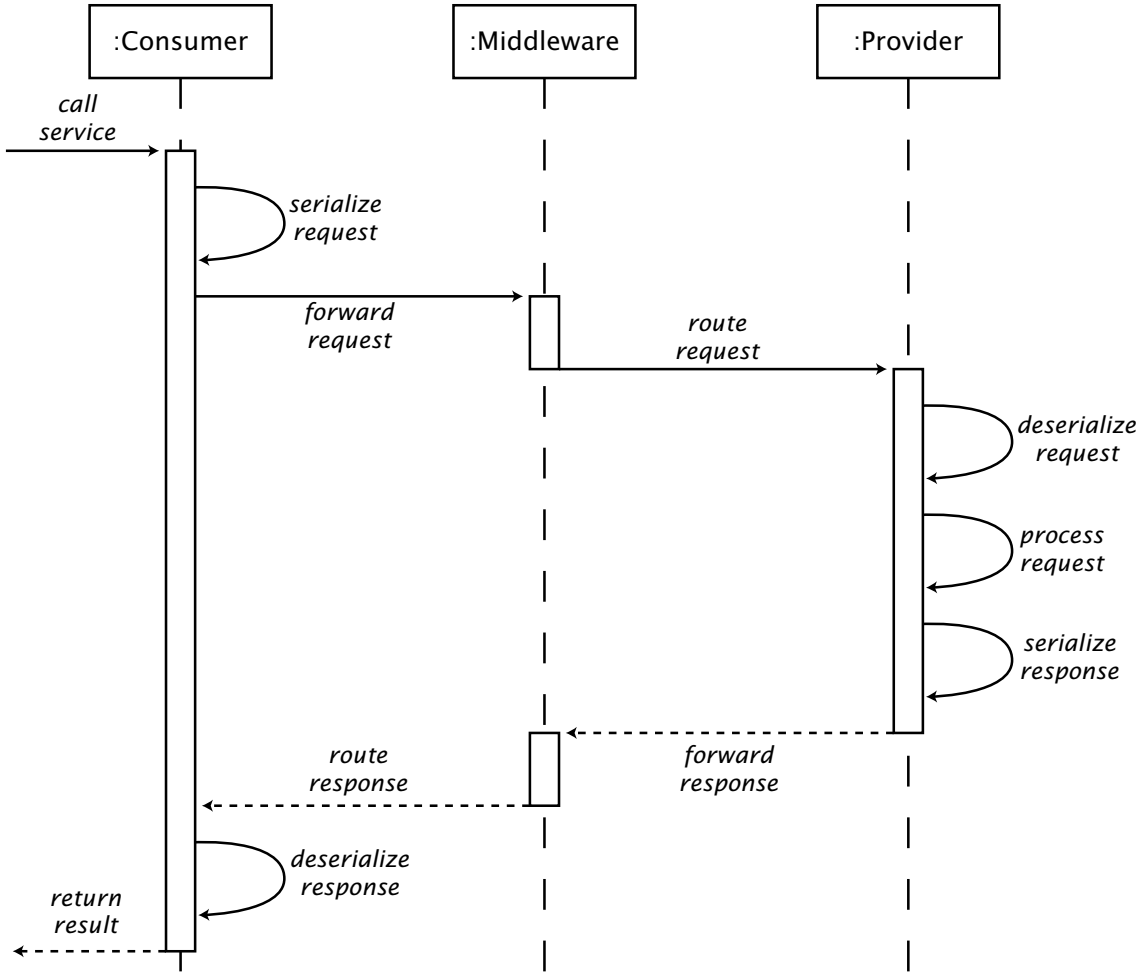
- OASIS standard
- *WS-Coordination* for managing conversational context
- *WS-AtomicTransaction* for short-lived activites with full ACID properties (two-phase commit)
- *WS-BusinessActivity* for long-running transactions with compensation

2 Performance

- two common reasons for systems not to live up to expectations: performance and security
- SOA about heterogeneity, so translations necessary
- SOA about distribution, so network latency an issue too
- how much do these matter?

(material from Josuttis, *SOA in Practice*)

2.1 Lifecycle of a service call



2.2 Serializing and deserializing

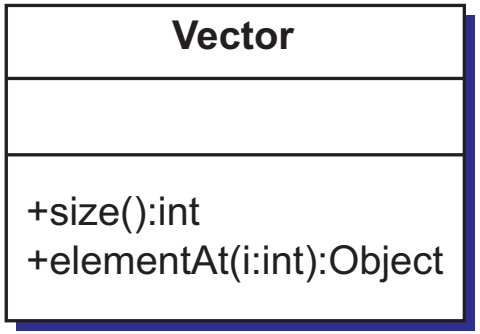
- necessary: cannot assume a common representation
- formatting, parsing and validation for XML messages take time
- increased bandwidth from chatty presentation (factor of 4 to 20 growth)
- message-level security aspects have an effect too

2.3 Transmission

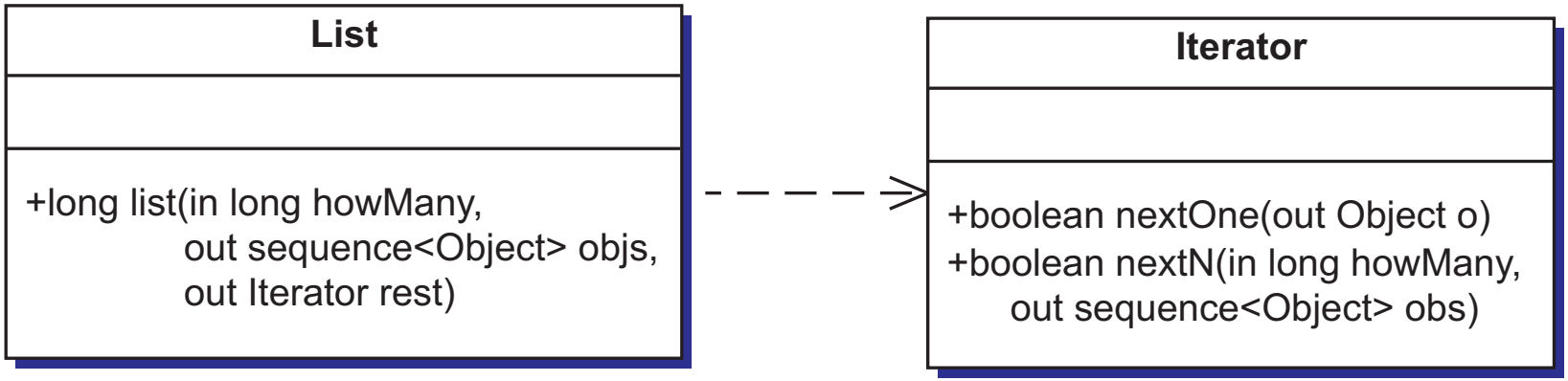
- in-memory call overhead about 100ns, service request about 100ms
- tempting to ignore network latency
- only possible for coarse-grained interactions
- service interface should provide convenient batched/chunked access
- (still awkward for client to use, though)
- treat automated OO–WS mappings with caution!

Example: Iterator

- Java *Vector* yields elements one at a time:

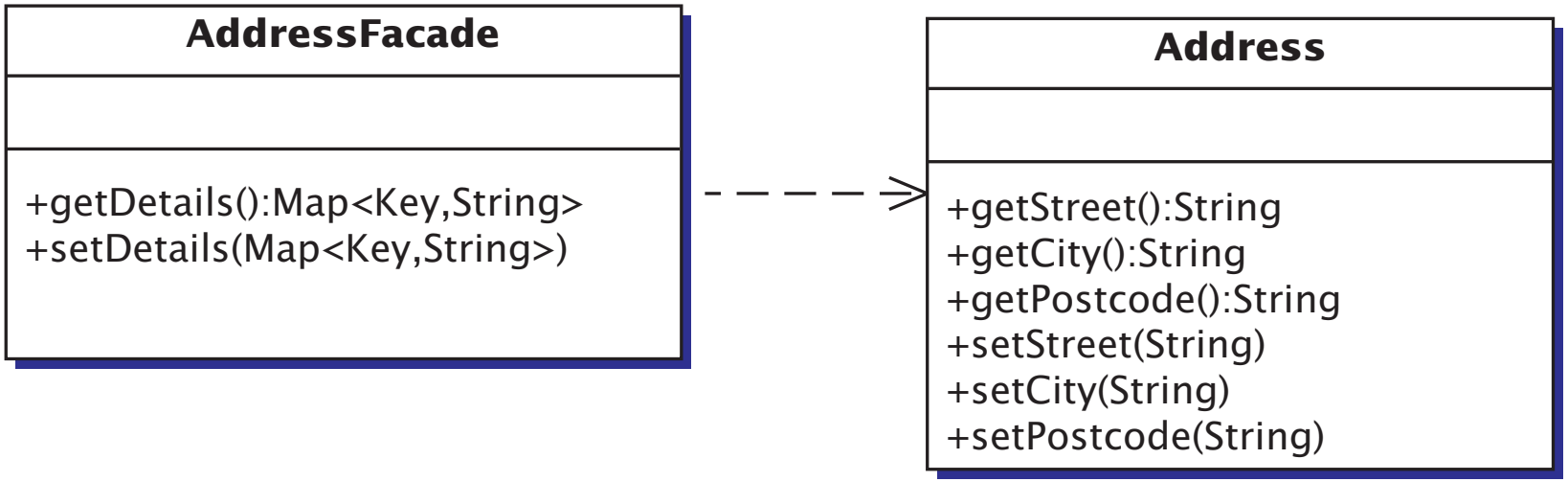


- typical distributed service is more careful — eg CORBA:



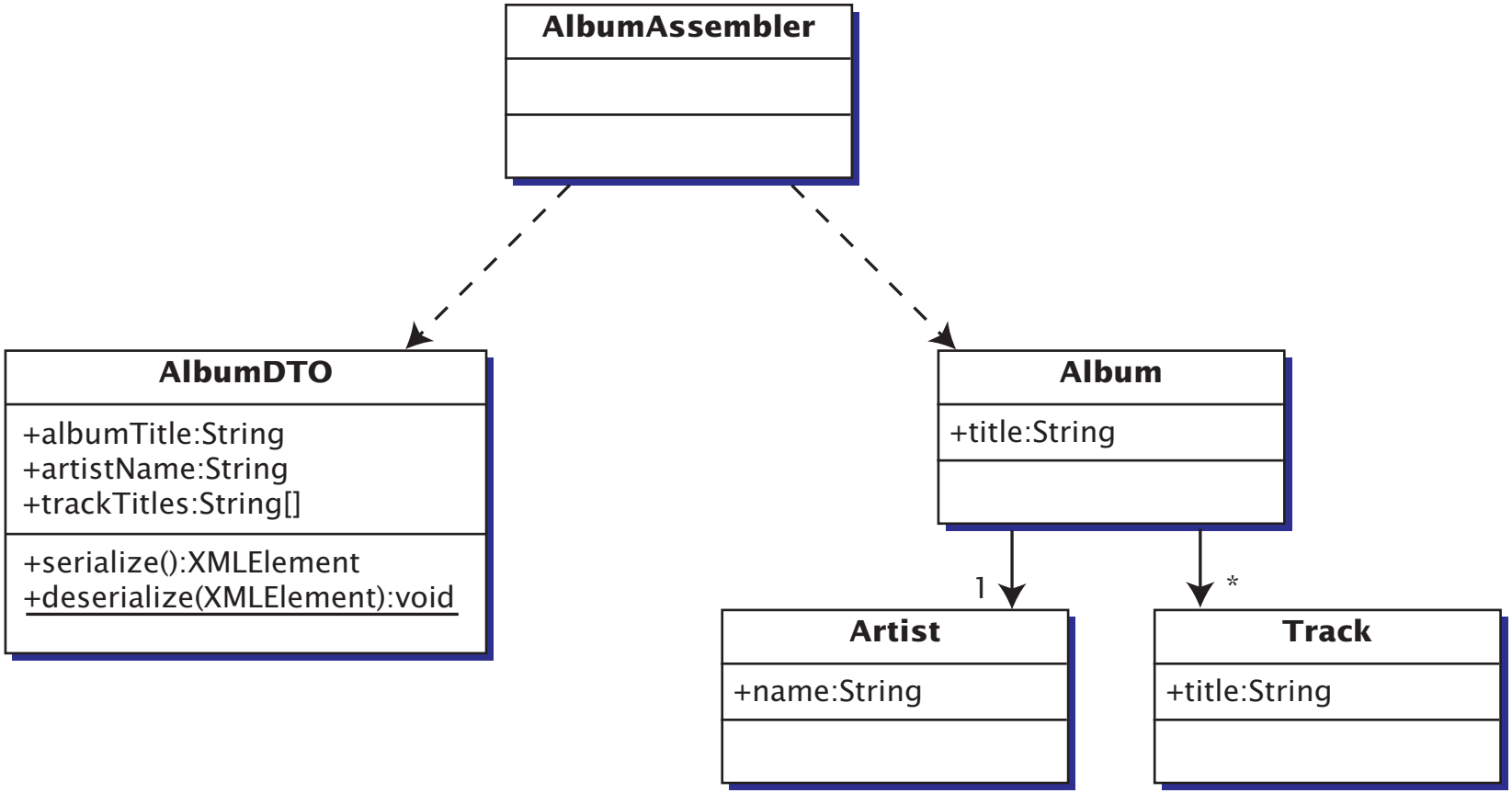
Example: Remote Façade (Fowler, PoEAA)

- *provides a coarse-grained façade on fine-grained objects to improve efficiency over a network*
- batched access to small methods
- fine-grained objects have no remote interface; façade has no business logic
- more obscure, more awkward: costs wrt productivity



Example: Data Transfer Object (Fowler, PoEAA)

- *an object that carries data between processes in order to reduce the number of method calls*



2.4 Distribution can't be transparent

The hard problems in distributed computing concern dealing with partial failure and the lack of a central resource manager. . . differences in memory access paradigms between local and distributed entities.

Many aspects of robustness can be reflected only at the protocol/interface level. . . An interface design issue has put an upper bound on the performance. . . Part of the definition of a class of objects will be the specification of whether those objects are meant to be used locally or remotely.

Jim Waldo et al, 'A Note on Distributed Computing', 1994

2.5 Processing

- major reason for slow service is processing time
- particularly an issue in the face of scaling
- service contract should *service level agreement* (SLA), specifying average response time and call rate
- might have to replicate service to achieve throughput
- if that doesn't work, need to decouple consumer (eg batching, asynchrony)
- as with all optimizations, worth profiling first!

2.6 Performance and reusability

- trade-off between conflicting forces
- latency encourages *coarse-grained* services (send one request rather than several)
- processing time encourages *fine-grained* services (don't send data that isn't needed)
- Josuttis proposes introducing special service for awkward consumer who wants an odd selection of fields (so much for reuse!)
- could have generic parametrized service, passed a *strategy*; but that has processing implications of its own

2.7 Performance and backwards compatibility

- another Josuttis war story: CRM system for phone company
- calls routed to operator; customer data preloaded to screen
- now company introduces an extra *status* attribute of customer (high status customers get routed to more experienced operators)
- simply another field returned by service: backwards compatible
- customer calls from one number, but may have multiple contracts
- so determine status from *total* volume of contracts
- but now have to load and process all contracts at run-time! doubled response time
- (solution: to compute status overnight; still expensive)
- moral: SLA is part of the contract too

3 Security

Classical notions of security are very relevant to SOA:

- *authentication* (who are you?)
- *authorization* (what may you do?)
- *confidentiality* (should you see this?)
- *integrity* (may you change this?)
- *availability* (are you being fair?)
- *accountability* (who is paying?)
- *auditability* (what happened?)

— few services will have no security requirements at all.

3.1 *Ad hoc* approaches to security

- just don't think about it
- just don't tell anyone your service is there
- just use firewalls to limit the service to the organisational intranet
- just roll your own

What do you reckon?

3.2 Means of security

- identification, authentication schemes
- authorization policies and services
- cryptography: encryption and signatures
- logging, logging services
- policy management

3.3 Basic technologies

- PKI (Public-Key Infrastructure)
- simple tokens
- cookies
- HTTP authentication
- SSL (HTTPS); also IPSec
- XML Encryption, XML Signature
- SOAP Signatures

Contrast *message-layer security* with *transport-layer security*.

3.4 PKI in a nutshell

- public-key encryption uses *key pairs*
- the encryption is asymmetric: one key *encrypts*, the other *decrypts*
- one key published: anyone can encrypt messages for me
- other key private: only I can decrypt
- signatures are dual: only I can sign, anyone can verify
- infeasible to reconstruct private key from public

Q: how do I verify that a key really belongs to a particular individual?

A: use a certificate that binds a public key to a name, via a signature from someone I already trust.

3.5 Simple tokens

- used by Google and Amazon Web Service APIs
- doGoogleSearch(*key*, . . .)
- token is set up out-of-band
- well-suited to a REST-style application

(+) easy to set up

(+) meets the service-provider's security goals (log/audit, availability management)

(–) no protection against impersonation

3.6 HTTP authentication

- **'basic version'**: username and password are base64 encoded and passed in an HTTP header

(+) implemented as standard

(–) not really much better than the simple token

- **'digest version'**: server sends a *nonce challenge*; response incorporates a digest value (MD5 hashes of nonce, username, password, other message data)

(+) much better; exposure is limited

(–) server must choose good nonces to prevent replay attacks; difficult to manage in 'session' contexts

(–) protects secrecy of password, but offers no other confidentiality

3.7 SSL/HTTPS

- strong authentication of server; optional strong authentication of client
- strongly encrypted two-way data channel

(+) standard and straightforward

(+) highly trustworthy

(–) entirely point-to-point

(–) no scope for processing by intermediaries (that includes firewalls and filters)

(–) protocol-based; no scope for fine-grained authorization

3.8 XML encryption

- use <EncryptedData>, <CipherData> to signal and contain the encrypted (base-64) data
- other tags record choice of algorithms, identity of key, etc
- very flexible: decide precisely what to encrypt; incorporate in your schema!
- good reading: *Why XML Security is Broken* by Peter Gutmann

3.9 Example

```
<?xml version='1.0'?>
  <PaymentInfo xmlns='http://example.org/paymentv2'>
    <Name>John Smith</Name>
    <CreditCard Limit='5,000' Currency='USD'>
      <Number>4019 2445 0277 5567</Number>
      <Issuer>Example Bank</Issuer>
      <Expiration>04/02</Expiration>
    </CreditCard>
  </PaymentInfo>
```

(Example from XML Encryption Standard, W3C, 10 Dec 2002)

Encrypt the `CreditCard` element

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
                          xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <CipherData>
      <CipherValue>A23B45C56</CipherValue>
    </CipherData>
  </EncryptedData>
</PaymentInfo>
```

(Eavesdropper can't even distinguish credit card use from money transfer.)

Encrypt finer-grained elements

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <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>
```

Credit card limit in the clear, but other elements (number, issuer, expiration) encrypted.

Encrypt the content

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <Number>
      <EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
        Type='http://www.w3.org/2001/04/xmlenc#Content'>
        <CipherData>
          <CipherValue>A23B45C56</CipherValue>
        </CipherData>
      </EncryptedData>
    </Number>
    <Issuer .../> <Expiration .../>
  </CreditCard>
</PaymentInfo>
```

Credit card and `Number` tag in the clear, but number itself encrypted.

Or encrypt the whole document

```
<?xml version='1.0'?>
<EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
 MimeType='text/xml'>
  <CipherData>
    <CipherValue>A23B45C56</CipherValue>
  </CipherData>
</EncryptedData>
```

Additional tags

```
<EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
        Type='http://www.w3.org/2001/04/xmlenc#Element'/>
   <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
   <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
     <ds:KeyName>John Smith</ds:KeyName>
   </ds:KeyInfo>
   <CipherData><CipherValue>DEADBEEF</CipherValue></CipherData>
</EncryptedData>
```

3.10 XML Signature

- more complex than encryption:
  - intimately tied to representation
  - *canonicalization* needed
- tampering with document, followed by removal of signature, may be invisible
- so design should take account of what to do with unsigned documents
- *detached signatures* used to allow whole document to be signed

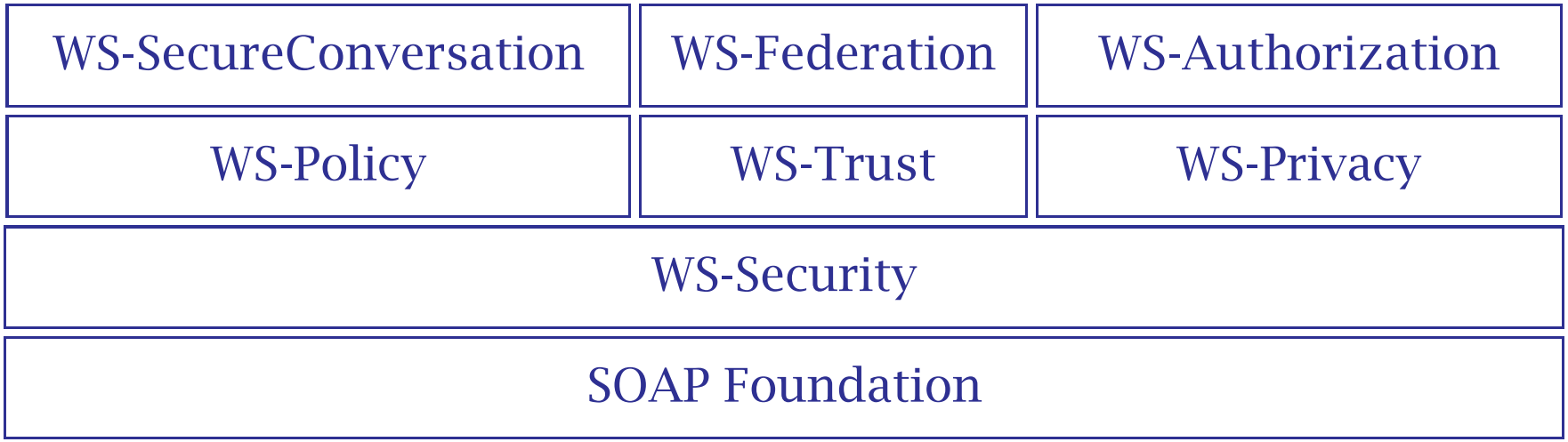
3.11 Example of detached signature

```
<Signature Id="MyFirstSignature"
     xmlns="http://www.w3.org/2000/09/xmldsig#">
 <SignedInfo>
  <CanonicalizationMethod
      Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
  <SignatureMethod
      Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
  <Reference
      URI="http://www.w3.org/TR/2000/REC-xhtml1-20000126/">
    <Transforms> <Transform
        Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
    </Transforms>
    <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
    <DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</DigestValue>
  </Reference>
 </SignedInfo>
```

```
  <SignatureValue>MC0CFFrVLtRlk=...</SignatureValue>
  <KeyInfo>
    <KeyValue>
      <DSAKeyValue>
        <P>...</P><Q>...</Q><G>...</G><Y>...</Y>
      </DSAKeyValue>
    </KeyValue>
  </KeyInfo>
</Signature>
```

(From *XML Signature*, W3C Recommendation 12 February 2002)

3.12 Web Services Security Roadmap



From IBM and Microsoft; see
www.ibm.com/developerworks/library/specification/ws-secmap/

3.13 Documents

(Varying levels of maturity. . .)

WS-Security: how to attach signature and encryption headers to SOAP messages; how to attach security tokens to messages (OASIS standard)

WS-Policy: capabilities and constraints of the security (and other business) policies on intermediaries and endpoints (OASIS submission)

WS-Trust: framework for trust models that enables Web services to interoperate securely; introduces notion of a *Security Token Service* (OASIS standard)

WS-Privacy: model for how Web services and requesters state privacy preferences and organisational privacy practice statements (vapourware)

WS-SecureConversation: how to manage and authenticate message exchanges between parties, including security context exchange and establishing and deriving session keys — long-running interactions, in contrast to those enabled by WS-Security (OASIS standard)

WS-Federation: how to manage and broker the trust relationships in a heterogeneous federated environment including support for federated identities (just a draft)

WS-Authorization: how to manage authorization data and authorization policies (vapourware)

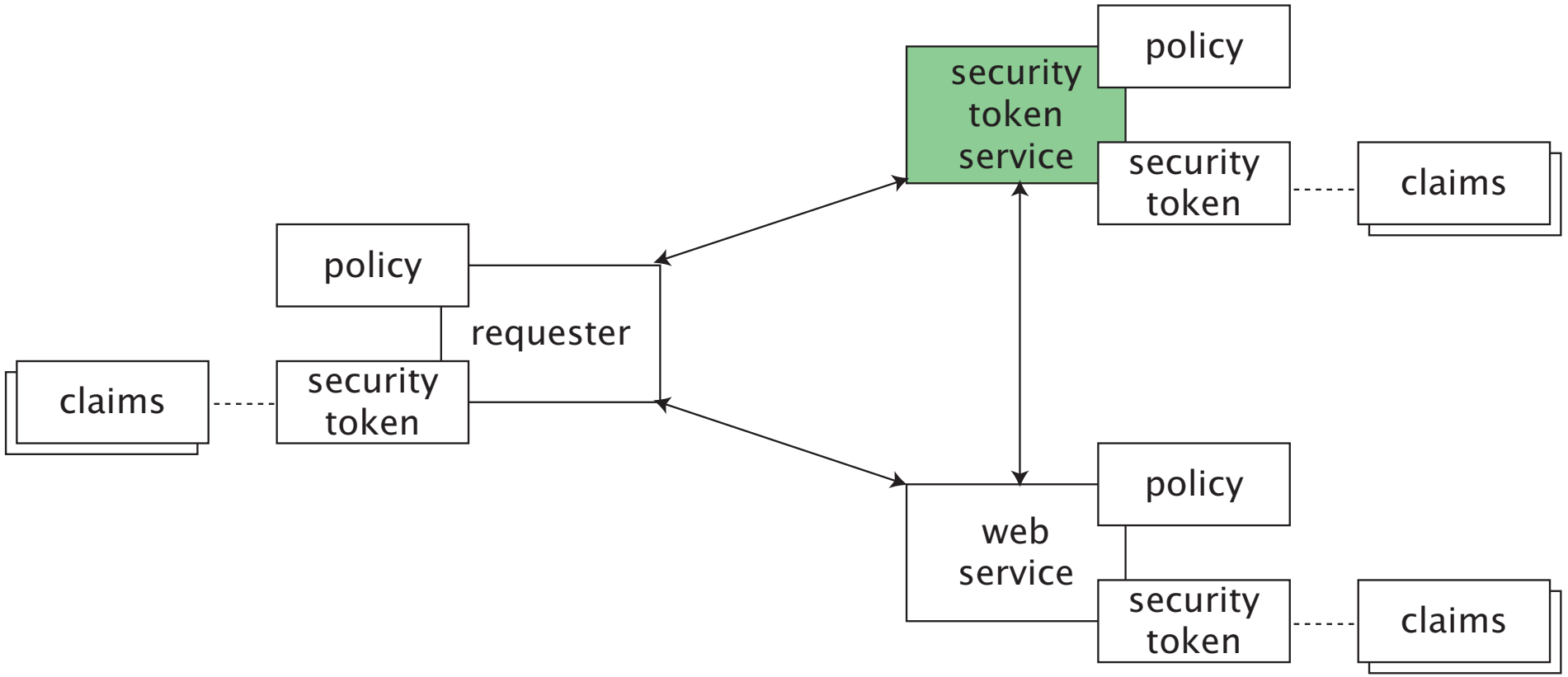
The roadmap also includes a number of instructive scenarios.

3.14 Terminology

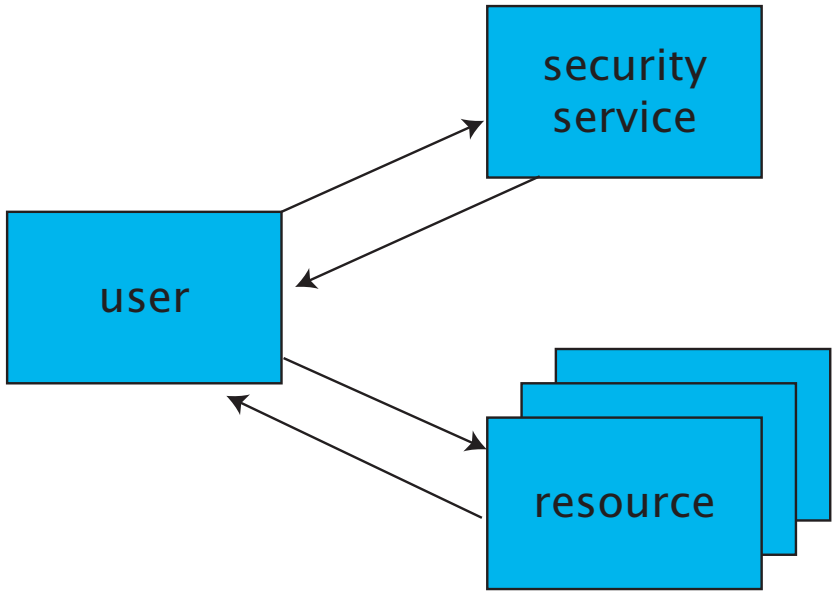
- security *token*, signed or unsigned
- *subject*: person, application, activity
- *claim*: made about the subject by the subject or another party
- *proof-of-possession* of token or set of claims
- *web service endpoint policy* for required claims
- *actor*: intermediary or endpoint (URI) (not user or client software)

3.15 Web Services trust model

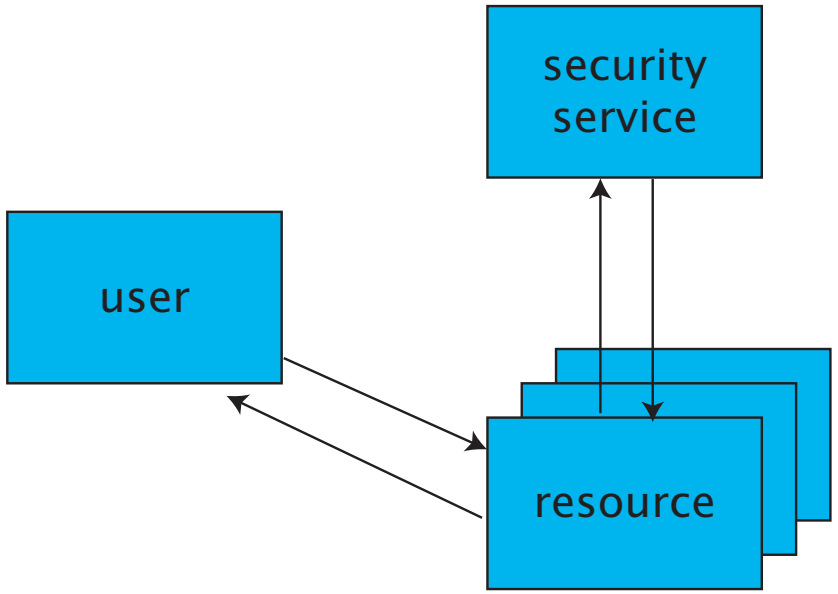
- service may have a *policy*, requiring requester to prove set of claims (e.g., name, key, permission, capability, etc.)
- requester can prove required claims by associating security tokens with messages
- when requester cannot prove required claims directly, it (or its agent) can try to do so by by contacting other *security token services* — which may in turn have their own policies
- security token services *broker trust* between different trust domains by issuing security tokens


policy
security token service
security token
claims
policy
requester
claims
security token
policy
web service
security token
claims

3.16 Push vs pull

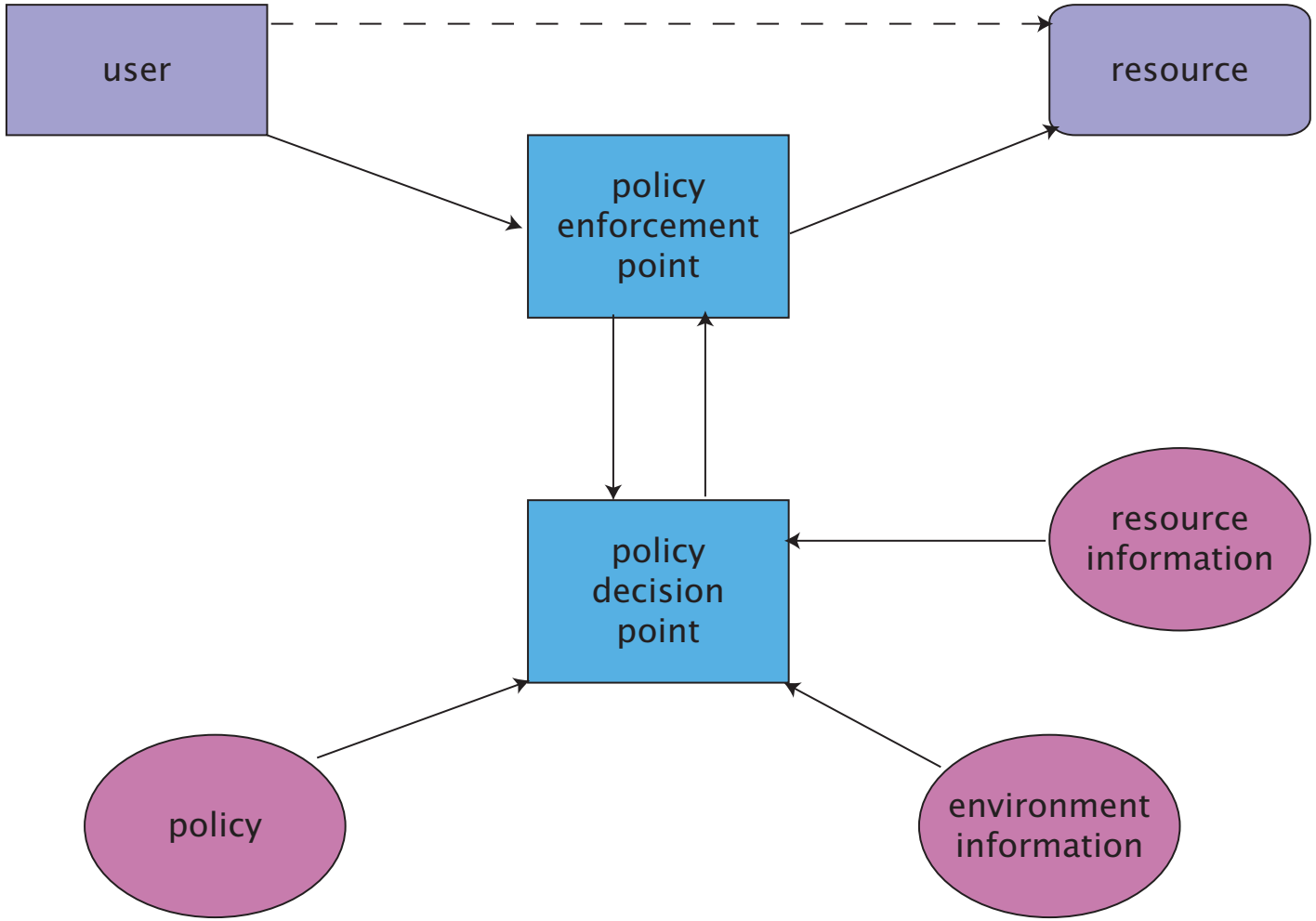


push model



pull model

3.17 Distributed decision-making



PEP asks PDP for a decision.

3.18 Consequences

- ‘difficult’ security decisions taken in one place
- resource can be in a separate domain from security service
- scalable in the number of users or resources
- push and pull models have different scalability characteristics
- scope for multiple security services, too (but inconsistency?)

Index

Service-Oriented Architecture

Monday	Tuesday	Wednesday	Thursday	Friday
Introduction Components	REST	Composition	Architecture	Engineering
coffee	coffee	coffee	coffee	coffee
Components	REST	Composition	Architecture	Conclusion
lunch	lunch	lunch	lunch	lunch
Web Services	Qualities	Objects	Semantic Web	
tea	tea	tea	tea	
Web Services	Qualities	Objects	Semantic Web	