CS144 – Nathan Tung (004-059-195)

Web page: TCP/IP (internet routing/transport), HTTP (communication for server/client & request/response), HTML (formatting/linking)
DNS translates logical name to IP, TCP/IP transmits bytes; MIME types/encoding tell type of data
HTML (structure): tags/text/comments; document = text+link+multimedia object
<!DOCTYPE html><html><head></head><body></body></html> <script type="text/javascript" src="script.js"></script>
<iframe> replaced <frameset> to embed content in document, hides multimedia incompatibility; allows dynamic page caching
CSS (style): specification for document formatting/presentation
<link rel="stylesheet" type="text/css" href="style.css"> <style type="text/css"> @import "example.css"; </style>
Separate structure from style: add same style for several pages, consistency, caching for unchanged CSS, make content independent for different mediums, hashing to tell when element changed
HTML5: added new tags <audio><video><canvas>, standardized API for JavaScript (events, canvas, web storage, drag-and-drop), rigorous error handling for browser interoperability
XHTML 1.1: more strict than HTML; tags/attributes must be lower case with matching end tags (even <p>) and quoted tribute values
HTML Forms: UI to submit user input to action (server destination) with methods GET (no side effects, encoded input in URL), HEAD (GET without input message), POST (encoded input in body, server-side changes), PUT (replaces location), DELETE (delete resource at URL), OPTIONS (request server option/info), TRACE (final recipient returns message as is; debugs proxy servers/server status)
<form action="http://www.google.com/search" method="GET"><input name="q" type="text"><input type="submit></form>
XML: standard tree-based model (DOM) of data with tagged/nested elements, element attributes, and text; single-root element, matching tags, unique attributes, starts with <?xml version="1.0" encoding="UTF-8">
Store XML: as RDB (better) or native XML
Query XML: XPath, XQuery
Specify XML schema: DTD, XML schema; use XML namespace to distinguish data/name conflicts
XML namespaces: applies to current tag and is children (not attributes unless explicitly prefixed)
DTD (Document Type Definition): written for SGML/HTML; grammar describing schema (element order/nesting and their attributes) in XML; order matters except for attributes; elements use parsed #PCDATA, attributes used CDATA
ID attribute in element can to pointed to with special IDREF attribute in another element
XML Schema: written for XML; more expressive than DTD
Use <sequence> for ordering; use targetNamespace="…" to specify ns for defined elements; use key/key-ref instead of ID/IDREFs
XPath: path expression navigates XML data (value of element is concatenation of all descending text node values); tree is NOT the same as XML DOM
Mixed Type Element: <Review> Great <title>WebApp</title> Class </Review> → XPath becomes /Review = Great WebApp Class → DOM contains 3 elements
XPath Symbols: beginning / is root; / means direct child; // means any descendant along path; * means any descendant; @ means attribute; [] means condition like [1] or [last()] or [@aaa]; . means specified node; | is union
MIME (Multipurpose Internet Mail Extensions): extends mail protocol to exchange different content (type/subtype: application/pdf, text/html/plain/xml, image/jpeg/gif/png, audio/mpeg/mp4, video/mp4/mpeg/avi, multipart/form-data/mixed/alternatives)
Data type specified by client or server in the HTTP tag like "Content-type: application/pdf"
H.264 (MPEG/LA) supported by most browser as de facto standard for video
Multiform-Part data: transmitting multiple info (say, to include multiple objects in a request) uses
<form … enctype=multipart/form-data …>; HTTP request can comprise multiple bounded parts
Binary data: simply tranmit as binary bytes; usually truncate first bit (8 bits to 7 bits) and assume ASCII; Base32/Base64/Quoted-Printable; Base64 = 2^6, so group binary string as 6-bit groups
Characters are stored/transmitted using a mapping "character encoding scheme": ASCII (7-bit, American, most popular), EBCDIC (8-bit, punch cards), ISO-8859 (8-bit, Latin-1), and other local/regional encodings
Asian languages used DBCS (Double Byte Code Character Set)
Code page (character encoding): unique number given to a character encoding by a system
Unicode: single unique number of every character independent from language/platform/program; 16-bit standard but now unlimited; characters map to a "code point" like A → U+0041 (41 00 big endian, 00 41 little endian)
UTF-16 scheme: wastes space, Unicode can't be handled by legacy apps even if only alphabet), needs entire string without error
Byte Order Mark: U+FEFF stored at beginning of string to tell endian mode (Big: FF FE, Little: FE FF)
UTF-8: de facto international standard and most popular, all ASCII chars mapped to single byte (English works with existing apps), recovery of string even if error
U+0000 ~ U+007F: 1 byte (0xxxxxxx) so ASCII maps to 1 byte!
U+0080 ~ U+07FF: 2 bytes (110xxxxx 10xxxxxx)
U+0800 ~ U+FFFF: 3 bytes (1110xxxx 10yyyyyy 10zzzzzz)
DB Normalization: Functional Dependencies
Charsets and Collations in MySQL
Character Set: collection of symbols and encodings; collation: set of rules for comparing characters in a character set
Web Server Architecture
Static content rom server: HTTP request delivers pregenerated content via a mapping from URL to server location
Dynamic pages: run program after parsing URL parameters, access DB to retrieve info from query, and return response
CGI (Common Gateway Interface), scripting pages (Apache+PHP, IIS+ASP), frameworks (Java Servlets, Node.js, Python+Django, Ruby on Rails, MS ASP.NET)
Web Site Layers: Encryption (encrypt transport) → HTTP (interpret request, serve response) → Application (generate dynamic content) → Storage/Data (store/retrieve data)
Storage Layer
File system: hard; need to avoid conflicts and inconsistencies; concurrency handling, slow data retrieval (ex: key-value pairs for user/password), security (ex: processes might have access to forbidden file records)
Relational database: (MySQL, PostgreSQL, Oracle, MS SQL, IBM DB2, Teradata); manage large datasets, SQL makes it easy to retrieve data, predefined schemas difficult to change, some data difficult to map to RDB
NoSQL DB: (Cassandra, MongoDB, FireBase, CouchDB); stores and retrieves data based on (key, value); emphasis on storing mechanism
Keyword-based query: keywords → relevant items; not supported by RDB or NoSQL; emphasis on retrieval mechanism
Information Retrieval (System): attempts to use given file info as knowledge
Boolean Model: ignore padding words (or, and, the) and find docs containing semantically important keywords; retrieve document based on whether or not it has all keywords
Inverted index: data structure (dictionary w/ posting or inverted list) that supports keyword-to-document ID mapping
Precision = |R&D|/|D| (what fraction of returned documents is relevant)
Recall = |R&D|/|R| (what fraction of relevant documents is returned)
Algorithm: for each document, extract all words into a list; for each word in that list, if it's not in dictionary, add it; then append the doc ID to the posting list for that word
Size estimate example: 100M docs, 10 KB/doc, 1000 unique words/doc, 10 bytes/word, 4 bytes/doc ID
Original data size: 100M docs * 10 KB/doc = 1 TB
Postings list size: 1000 words/doc * 100M docs * 4 bytes/doc ID = 400 GB
Dictionary size: unique total words is about sqrt(# docs), so about 1 M; 1 M words * 10 bytes/word = 10 MB
Inverted index size is the bottleneck; it's roughly equivalent to corpus size; if inverted index size > memory, we'll need to merge smaller ones based on keywords
Vector Space Model: weighing documents based on relevance using tf (term frequency) and idf (inverse document frequency)
tf.idf = f*log(N/n), where tf: # times term t appears in doc D, idf: total # of documents/number of documents with term t
Cosine similarity: dot product of two vectors to compute similarity of docs to query (using the tf-idf vector)
$cos\theta = \overline{doc}\ \overline{Q}/|doc||Q|$, angle between doc terms and query; divide by doc size to standardize score; divide by query to get cosine
Topic-based indexing: index docs on concepts rather than terms; given query keyword, find best matched topic; use topic to search index
Latent semantic indexing (LSI): approximate nxm matrix, SVG on doc-term matrix, keeping top-k singular values (M=D S T), represent documents by doc-concept matrix (compute similarity of docs based on that matrix)
Probabilistic topic model (Latent DIrichlet Analysis, or LDA): probabilistic doc generation model and probabilistic topic inference; works!
Link-based ranking: tf-idf doesn't work in some situations, like picture based text/sites, so we use links from other pages (based on that page's credibility) (with keywords) to rank relevance
PageRank: PR(pi) = PR(p1)/c1 + PR(p2)/c2 + … + PR(pk)/ck
where p1…pk are pages that have a link to pi, and c1…ck are the # of links in that page
Random-surfer model: probability that user who randomly browses the web and clicks link starting from some page will end up at page i
Dangling page: problem when values dissipate through pages without any links
Crawler trap: pages that point to each other; all values may accumulate on those pages
Spatial Indexing: query based on locations
Range query: given a range, return set of points within range
Nearest-neighbor query: return closest point to given point
Where-am-I query: given a point, which object overlaps with point
Partial-match query: given value for a specific dimension, return all points matching value in that dimension
Example: 1M points within 0<x<100 and 0<y<100, uniformly distributed, with 100 points per disk block. Indexing?
1M/(100 points/block) = 10,000 bocks to retrieve if we cluster points based on a primary "clustered" index; secondary index doesn't help too much
Grid File: space partitioned into grid; in each dimension, gridlines partition space into stripes
Quad Tree: region cursively partitioned into 4 equal-size quadrants until all points in region can fit into a single disk bucket
R-Tree (Region Tree): each internal node of R-tree contains a set of (arbitrarily-sized) subregions as children
data points stored at leaf nodes
Web Services
SOAP (Simple Object Access Protocol): standard for making request and receiving response
Root element is <Envelope>, which wraps a child <Body>, which wraps the soap message
WSDL (Web Service Definition Language): standard for describing a certain Web service, providing location, list of operations, and input/output parameters
Types: format of I/O parameters; Message: message structure; PortType: list of operations and parameters; Binding: binding protocol and encoding scheme (operation I/O scheme) ; Service: the location (URL)
REST (Representation State Transfer): request encoded in URL, response encoded in simple XML (no messaging layer) or in JSON
Lightweight, human readable, easy to build, more popular
SOAP is rigid (type checking) and has automatic development tools
Web Service vs. RPC: web service is XML based, uses HTTP, and thus has fewer compatibility issues
Distributed Transactions: travel site handles flight and hotel; how is booking handled, what if CC fails or flight/hotel no longer available?
No one can commits unless everyone commits
Two-phase commit: before committing, ask everyone else if they're ready
If everyone says yes, commit (anyone who says ready cannot back out); if anyone says no, abort
Asynchronous Transaction: each participant commits at own pace; transaction is a sequence of smaller transactions by each participant (until everyone has committed)
Must remember compensation transaction to "rollback" a committed transaction just in case someone doesn't commit; dependencies of transactions must be tracked
Use two-phase commit when some people aren't willing to participate or commit
Use async transaction when participants are fast and willing to commit
Remembering State and Persistent Offline Storage:
Use cookies; servers ask for non-expired relevant cookies from browser in HTTP request; never leaked over cross-domain servers
document.cookie = 'ppkcookie1=testcookie; expires=Thu, 2, Aug 2001 20:47:11 UTC; path=/'
localStorage (key-value store in HTML5, permanent and remembered) vs. sessionStorage (per page, temporary)
Security: DON'T TRUST USER INPUT
Defacement (DDoS), SQL/command injection, spoofing/phishing, MITM, buffer overflow, pharming/DNS poisoning, password brute-forcing, packet sniffing, client-state manipulation, cross-domain (XSRF, XSSI, XSS)
We want **CIAA (Confidentiality, Integrity, Authentication, and Authorization)**
Confidentiality: steganography (hide true message by obscuring it in something else), encryption (scramble message with key)
Symmetric Key Cryptography (DES, AES)
Shannon Perfect Secrecy: knowing the cipher doesn't affect the probability of getting the plaintext P(m|c) = P(m)
OTP (one time pad/password) encryption is perfectly secret, but impractical
Requires n(n-1)/2 keys for n parties, but cheap to use
c = F(m, k) and m = F'(c, k) where F and F' generate ciphers
Asymmetric Key Cryptography (RSA)
F'(F(m,e),d) = m
Perfect Secrecy: c = F(m,e) cannot obtain m
e cannot obtain d
Requires 2n keys for n parties, but expensive to use (
Confidentiality: use other party's public key to encrypt; then use symmetric cryptography to continue
Authentication: generate random value r (nonce), encrypt it with their public key; they need to send back the same value
Integrity: use checksum/hash h, sign it with your private key and send it; they decrypt it using your public key and then compare
Validating public key? Use certificate authority (CA) out-of-band identity check to vouch for their public key, grant a certificate
RSA Algorithm:
Pick two random prime numbers p and q; pick e < (p-1)(q-1); find d<(p-1)(q-1) such that d(e mod (p-1)(q-1) = 1
Theorems: there exists such a unique d if e is a coprime to (p-1)(q-1) (does not share factor with that product); assuming n=pq, m=m^(ed) mod n
RSA Problem: F(m,e) = c = m^e mod n
Large Number Factorization Problem: solving for d in d*e mod (p-1)(q-1) = 1
RSA is 1000x slower than symmetric cryptography; use asymmetric only to agree on symmetric keys at the beginning
SSL: client contacts server, who presents signed certificate showing public key; client authenticates, agrees to exchange symmetric key via public (asymmetric cryptography); communicate securely
Key Management
Need perfectly random number generator and "safe" key storage (USB drive, not so good; smartcard is tamper-resistant; OTP card prevents password replays)
Check by what you have (physical key, id), what you know (password), who you are (biometrics like fingerprint)
2-factor authentication
Buffer Overflow: not checking for user input length might overwrite overflow/overwrite return address with an "attack string"
Stackguard: inserts random "canary" before return address and checks if corrupted before attempting to return; if unchanged, there's a low chance of attack
Sandbox: limits attacker to minimal privileges
Sendmail used to run with root privileges, attracted lots of hackers
Client-state manipulation: clients can change their info via cookie poisoning
Maintain authoritative state at server, only give session id (to prevent stolen ID, use random number from a large pool; use client IP to generate it; make it expire quickly)
That message with session ID can either be encrypted/decrypted on use; but this is expensive and requires back-end storage
Or use a checksum to see if the parameters from client match signature (that we signed on issue) (must sign every parameter; make signature state expire; this prevents price fluctuations, or changing item name, etc.)
SQL Injection: attacker can make a statement always true to return sensitive info
Use minimal privileges, sanitize user input; use prepared statements; "taint" support in Perl/Ruby; encrypt sensitive database info
Cross Site Scripting (XSS): a page that accepts user input for long periods of time without sanitizing might execute scripts; escape the input; use white listing (safer than black listing)
Cross Site Request Forgery (XSRF): when cookies are stored long term (future expiration, permanently on HD), malicious page can submit a form to another site
Even with same-domain/same-origin policy, site won't reject our form submit/request since we're authenticated through cookie
Same-domain policy restricts attacker to writing to server, not reading
To prevent, use "action token", a secret-key signed session ID that's random, unique, and short lived; embed this in hidden field on form; each request needs to compute and compare action token with session ID
Scalability
Possible bottlenecks: Disk/DB IO, network, or CPU/memory
Estimation: serving a website from HDD to internet to web page (1000 Mbps =~ 100 MB/s), webpage is 10 KB
Internet: 100 MB/s*1 page/10KB = 10,000 pages/s
HDD: 100 pages/s, since 100 IO/s is the bottleneck; unless our data is sequential, then our bottleneck is HDD speed; at 100 MB/s, we get the same 10,000 pages/s
Profiling tools: top (CPU/processes), iostat (disk io and tps: transactions per second, like disk IOs), netstat (network io), etc.
How to scale a web site?
Scale up (buy more expensive server): easier, less software changes, has upper limit, expensive
Scale out (add more machines, preferred method today): cheaper, no upper limit
Web Server Architecture
Load Balancer (TCP NAT request distributor; hardware (expensive/powerful), software, DNS round robin) to manage access to layers
Encryption (transport encryption, SSL): easy to scale out, run in parallel (linear relation between #requests and #cpu)
HTTP Server (apache)
Application Server (tomcat): take requests, go to storage, combine results, give back response to http layer
Defer concurrency issues to storage layer so app layer becomes linear
Storage (persistence layer, MySQL)
Add ram to scale database for memcaching
Global read only: replicate underlying DB over many machines; no sync problems
Local read, local write: partition/shard data from users into machines; on access, user ID gets hashed and assigned a single machine with their data; no sync problems
Global read/write: not linearly scalable by replicating or partitioning; very challenging; companies buy larger scaled-up machines
Example: 2:1 read and writes/user/second; given that we get 30 IO/s/machine, that machine can serve 30/(2+1)=10 users
What about 20 users? 40 reads (distributed any way) and 20 writes (per machine!) means 20 reads+20 writes=40 IO; this needs NOT 2 machines, but 4…
NoSQL: RDBs are very limited; expensive data retrieval, powerful/expressive but complex and hard to scale out
We don't actually need **ACID** (**Atomicity, Consistency, Isolation, and Durability**) but it's nice
Byzantine generals problem models ACID and perfect synchronization
Two generals A and B need to attack together to win, but they're separated; messengers get lost with some probability
If A sends a messenger to B and attacks, the messenger might not have made it; same with all the ACKS we send…
Send multiple messengers; keeping states on multiple nodes synced is very costly
CAP Theorem: Consistency (after an update, all readers can see it), Availability (continuous operation despite node failure), Partition Tolerance (continuous operation despite network partition)
Only two of three here can be guaranteed
ACID vs BASE (Basically Available, Soft-state – not always consistent, Eventual consistency – but will be eventually)
Read Your Own Writes (RYOW) consistency shows your own updates
Session Consistency maintains consistency within sessions
Monotonic-read consistency shows latest update; on next access, we always only see that update or later
(**key, value) store**: inefficient read and update for single-fields
Column-oriented store: unique row-key and some columns (column families preserve locality) allow more efficient update/retrieval, but still based on row-key
Document datastore: documents have fields that are (key, value) pairs; user can build index on some fields, allows data retrieval on those non-key fields; more complex, less scalable
Consistent hashing (partitioning): hash objects/nodes to hash ring; new added nodes only require 1/N (N=#nodes) redistributions
Treat nodes as (say) 10 virtual nodes mapped on ring; then take 10 nodes and put data into 1 physical node, etc. to minimize imbalance when we have few nodes
Cluster-based architecture: hardware failures from power supply (power/heat issues), hard drive (last 1 year, but claims 3), and network
Large Scale Distributed File System
SAN (Storage Area Network): scale up approach for disks; nodes in cluster share one big disk; convenient but expensive
Distributed File System (network – NFS) (GFS): filesystem with cluster of machines (despite node failures); split data into chunks distributed to nodes; replicate to multiple nodes; master server remembers filename to chunk mapping

REQUEST

```
GET /cs144/example.html HTTP/1.1
Host: oak.cs.ucla.edu (name of web server; virtual hosting)
User-Agent: Mozilla/5.0 (info on client software)
Referrer: http://oak.cs.ucla.edu/cs144/ (page linking to request)
Accept: text/xml, text/html, … (media/content acceptable)
Keep-Alive: 300
Connection: keep-alive (keep 300s connection for multiple requests)
Cookie: … (adds state)
```

RESPONSE

```
HTTP/1.1 200 OK
Date: Wed, 04 Apr 2007
Server: Apache
Last-Modified:
ETag: “15b64b-af-ebdb-940”
(unique tag depending on content, for caching)
Content-Type:
```

DTD

```
<!ELEMENT Bookstore (Book*, Author*)>
<!ELEMENT Book (Title, Remark?)>
<!ATTLIST Book ISBN ID #REQUIRED
                Price CDATA #REQUIRED
                Edition CDATA #IMPLIED
                Authors IDREFS #REQUIRED>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Remark (#PCDATA)>
<!ELEMENT Author (#PCDATA | (First_Name, Last_Name))>
<!ATTLIST Author Ident ID #REQUIRED>
<!ELEMENT First_Name (#PCDATA)>
<!ELEMENT Last_Name (#PCDATA)>
```

Java Function

```
public class Converter {
    public double fahrenheitToCelsius(double fahrenheit) {
        // convert Fahrenheit to Celsius
        return (fahrenheit-32.0)*5.0/9.0;
    }
}
```

SOAP Request

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
    <soap:Body>
        <fahrenheitToCelsius>
            <fahrenheit>80</fahrenheit>
        </fahrenheitToCelsius>
    </soap:Body>
</soap:Envelope>
```

SOAP Response

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
    <soap:Body>
        <fahrenheitToCelsiusResponse>
            <fahrenheitToCelsiusReturn>26.67</fahrenheitToCelsiusReturn>
        </fahrenheitToCelsiusResponse>
    </soap:Body>
</soap:Envelope>
```

ANSWER:

```
<?xml version="1.0"?>
<Products xmlns="http://cs144.ucla.edu">
<Book ISBN="1423902017">
<Title>Web Application</Title>
<Author>John Cho</Author>
<Edition><Ed>1</Ed><Year>2008</Year></Edition>
</Book>
<CD AISN="B001GO09MI">
<Title>Circus</Title>
<Artist>Britney Spears</Artist>
<Review>She is BACK!!!</Review>
<Review>No, it is not a comeback, she never left us!</Review>
</CD>
</Products>
```

ANSWER:

```
<!DOCTYPE Products[
<!ELEMENT Products (Book*, CD*)>
<!ELEMENT Book (Title, Author, Review*, Edition+)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Author (#PCDATA)>
<!ELEMENT Review (#PCDATA)>
<!ELEMENT Edition (Ed, Year)>
<!ELEMENT Ed (#PCDATA)>
<!ELEMENT Year (#PCDATA)>
<!ELEMENT CD (Title, Artist, Review*)>
<!ELEMENT Artist (#PCDATA)>
<!ATTLIST Book ISBN CDATA #Required>
<!ATTLIST CD AISN CDATA #Required>
]>
```

WSDL

```
<?xml version="1.0"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://oak.cs.ucla.edu/cs144/Converter"
             xmlns:tn="http://oak.cs.ucla.edu/cs144/Converter">
    <!-- types element specifies the "type" of input/output parameters -->
    <types>
        <xs:schema targetNamespace="http://oak.cs.ucla.edu/cs144/Converter">
            <!-- fahrenheightToCelsisus will be the root request element -->
            <xs:element name="fahrenheitToCelsius">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="fahrenheit" type="xs:double"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>

            <!-- fahrenheitToCelsiusReponse is the root response element -->
            <xs:element name="fahrenheitToCelsiusResponse">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="farenheitToCelsiusReturn" type="xs:double"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:schema>
    </types>

    <!-- message element specifies request and response message format message element
may consist of multiple parts like header, body, faultcode, etc-->
    <message name="convertRequest">
        <part name="requestBody" element="tn:fahrenheitToCelsius"/>
    </message>
    <message name="convertResponse">
    <part name="responseBody" element="tn:fahrenheitToCelsiusResponse"/>
    </message>

<!-- portType element specifies the available operations (= methods)
     and the associated input/output message format -->
<portType name="ConverterPortType">
    <operation name="fahrenheitToCelsius">
        <input message="tn:convertRequest"/>
        <output message="tn:convertResponse"/>
    </operation>
</portType>

<!-- binding element specifies the transfer protocol and the message MIME encoding of the
service every operation specified here should have a corresponding operation in the
associated portType -->
<binding type="tn:ConverterPortType" name="ConverterBinding">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="fahrenheitToCelsius">
        <soap:operation soapAction=""/>
        <input>
            <soap:body use="literal"/>
        </input>
        <output>
            <soap:body use="literal"/>
        </output>
    </operation>
</binding>

<!-- service element specifies the URL where the service is available -->
<service name="ConverterService">
    <port binding="tn:ConverterBinding" name="Converter">
        <soap:address location="http://oak.cs.ucla.edu/cs144/Converter"/>
    </port>
</service>
</definitions>
```