Technologies for Web and Cloud Service Interaction: A Survey

Harald Lampesberger

Abstract The evolution of web and service technologies has led to a wide landscape of protocols and standards for interaction between loosely-coupled software components. Examples range from web applications, mashups, apps, and mobile devices to enterprise-grade services.

Cloud computing is in a sense the industrialization of service provision and delivery, where the web and enterprise services are converging on a technological level. The article discusses this technological landscape and, in particular, current trends. The survey focuses on the communication aspect of interaction by analyzing languages, protocols, and architectures that drive today’s standards and software implementations applicable in clouds.

Technological advances will affect both the client- and service-side. In particular, there is a trend towards multiplexing, multihoming, and encryption in upcoming transport mechanisms, especially for architectures, where a client simultaneously sends a large number of requests to some service. Furthermore, there is a trend towards client-to-client communication capabilities in web clients that has the potential to establish a foundation for novel web-based messaging architectures.

Keywords Web technology · Web services · Cloud services · Service architecture · Communication protocols · Languages · Service interaction patterns

1 Introduction

In the past decades, we have seen two evolutions. The World Wide Web has emerged from a simple network of hypertext media documents to mashups, interconnected social platforms, and mobile device compatibility. On the other hand, the need for scalable enterprise services in heterogeneous ecosystems has led to numerous middleware standards and service-oriented architectures (SOA) [194] for distributed computation. The two evolutions are converging; while web applications have begun to integrate service concepts for scalability and flexibility, enterprise services resort to web standards and protocols to reach a large number of platforms and devices over the Internet.

Cloud computing [23] can be seen as the industrialization of service provision and delivery over the Internet using established technologies. The cloud accelerates the convergence of the web and services worlds. A shared goal is to offer some form of service, accessible across devices, systems, and platforms. Interaction by communication between service consumers and providers, i.e. clients and services, is therefore a key aspect of service delivery.

Web technology has become pervasive and is not limited to hypermedia applications anymore. Standards are widely accepted and they have contributed to the success of web services because protocols such as the Hypertext Transfer Protocol (HTTP) are reliably forwarded over the Internet [4, 232]. Cloud service delivery models such as Platform-as-a-Service (PaaS) or Software-as-a-Service (SaaS) benefit from the wide acceptance of web standards [23, 268]. To acknowledge this relation between web and services technology, this article surveys the state-of-the-art and recent trends in technologies that are applicable in clouds.

1.1 Scope

A survey of technologies in such a dynamic environment like the web needs a defined scope. All technologies that allow a client to interact with a service should be consid-
ered, but nevertheless the notion of service is not clearly defined [268]. The following informal properties and restrictions are therefore stated to characterize a service in context of this work:

– **Service interface.** Services are considered as distributed network-accessible software components that offer functionality and need communication for interaction [96, 232]. A notion of interface that accepts a certain language is therefore required. The survey is restricted to technologies that enable communication between clients and service interfaces applicable in web, PaaS, and SaaS cloud delivery models.

– **Heterogeneous platforms.** A characteristic of service-orientation is the provision of functionality and content across hard- and software platforms. Only technologies that embrace this compatibility are considered.

– **Publicly Available Standards.** The focus is on technologies that are available to the public audience, in particular, technologies based on Internet protocols, i.e. the TCP/IP protocol suite [276], and with publicly available specification. Specialized technologies for a limited audience or application, like industrial control systems, are not part of this study.

– **Parties.** There are two participating parties or peers in service interaction: a client that consumes some service offered by a provider or server, i.e. client-to-service interaction. On a conceptual level, a service can also take the role of a client to consume other services in a composition, i.e. service-to-service interaction. Furthermore, a service can signal between two clients to establish client-to-client or peer-to-peer interaction.

In accordance to the aforementioned characteristics, the state-of-the-art and recent trends in web and service communication technologies applicable to the web, PaaS, and SaaS are surveyed.

### 1.2 Motivation

This survey is motivated by ongoing research efforts in formal modeling of cloud services [42, 38, 268], modeling of service quality [246, 245], service adaptation [48], identity management [299, 298], and security monitoring [145, 144, 146] for web and cloud services. All these aspects assume communication between clients and services. Understanding the state-of-the-art in service communication is therefore necessary, especially for security research because an ambiguous or imprecise service interface is in fact a gateway for attacks [265].

There exists a rich body of literature using patterns to describe service interaction on a conceptual level [1, 25, 24, 106, 349]. On the other hand, the numerous software implementations used in today’s services are heavily driven by continuously evolving standards and ad-hoc specifications. This work aims to bridge this gap by surveying the state-of-the-art of technologies and resort to patterns when concepts are discussed. Patterns are appealing because they allow to describe solutions in a conceptual way and can therefore support service integrators and scientists in understanding new technologies. The relations between protocols, architectures and service interaction patterns are also bidirectional. On one hand, patterns have been derived from successful protocols, on the other hand, patterns have influenced the specification of new protocols.

### 1.3 Methodology and Structure

The problem is approached both in top-down and bottom-up manners. The top-down view investigates what entities of information are communicated between clients and services, i.e. how languages can encapsulate and transport content and media. The bottom-up view discusses communication protocols and architectures for actually exchanging content between clients and services. Figure 1 visualizes the relationships between concepts required for service interaction.

*Fig. 1* The concepts and their relationships found in communication technologies also reflect in the structure of the survey.

*Languages* are fundamental for communication. A language defines an alphabet, syntax, and semantics to represent information in a transportable format. Languages therefore encode content, media, or information in general. Popular languages in the web and for services are discussed in Section 2.

Due to the many forms of interaction, conceptual patterns for a unified nomenclature are considered, in particular, *service interaction patterns* (Section 3) introduced by Barros, Dumas, and Ter Hofstede [24, 25] in context of the Workflow Patterns Initiative [343]. These patterns are rather informal but manageable in their number and sufficiently abstract to discuss interaction in the survey.

A *protocol* specifies a language and rules of engagement between communicating parties. Protocols can integrate other protocols to become a protocol stack as commonly seen in Internet protocols. Relevant protocols for web and cloud services are surveyed in Section 4.
Architectures integrate protocols and languages for service delivery through communication. An architecture is basically a blueprint for service delivery and Section 5 investigates common architectures found in the web and services applicable to cloud computing. Architectures are eventually implemented as executable software and popular implementations are discussed when appropriate. Scientific findings, observations, and potential implications are then discussed in Section 6, whereas Section 7 concludes the survey. The contributions are:

1. An overview of text-based, binary, and container formats to encapsulate information in a transportable format;
2. An overview of modern protocols in terms of the TCP/IP protocol stack, including multiplexing and multihoming transport mechanisms, HTTP extensions, and wire formats for messaging protocols;
3. A survey of architectures distinguishes between a web-oriented view (i.e. web applications, web syndication, and web mashups) and a service-oriented view (i.e. remote procedure calls, web services, and messaging solutions); and
4. A discussion of observations, in particular: multiplexing, multihoming, and encryption in modern transport mechanisms; correctness of content types; upcoming client-to-client capabilities; and the impact on network traffic monitoring.

1.4 Standardization Bodies

Standards for languages, protocols, and architectures in the Internet, web, and for services are primarily driven by non-profit organizations, communities, consortia but also enterprises. Important institutions are therefore recalled.

To develop industrial standards on a global scale, including specifications for electronic communication devices, e.g. networking, the International Organization for Standardization (ISO) [118], the International Electrotechnical Commission (IEC) [117], and the International Telecommunication Union (ITU) [119] are three connected organizations that closely work together. The Institute of Electrical and Electronics Engineers (IEEE) Standards Association [116] is also well-known for global networking standards, e.g. the IEEE 802.3 Ethernet standard.

With respect to language and protocol specifications, the Internet Engineering Task Force (IETF) [122] organizes a community process to develop standards, especially communication protocols, through open Request for Comments (RFCs). The World Wide Web Consortium (W3C) [344] is renowned for standardization of web protocols, and the Object Management Group (OMG) [207] aims for business processes and modeling standards. Another non-profit organization for developing open standards for languages and protocols, specifically for enterprise services, is the Organization for the Advancement of Structured Information Standards (OASIS) [220].

The Internet Corporation for Assigned Names and Numbers (ICANN) [121], including its department Internet Assigned Numbers Authority (IANA) [120], is a non-profit organization for directing world-wide the agreement on Internet addresses, domain names, and protocol identifiers. Also, a number of standards have been proclaimed by enterprises that use them internally or offer them as software or services. Some examples are Amazon, Cisco Systems, Google, Facebook, IBM, Microsoft, and Oracle.

2 Languages for Content and Media

Formally, a language is a (possibly infinite) set of strings generated from a finite set of symbols, referred to as alphabet [107]. Languages are essential to communicate information in form of messages. While information exchange in web and cloud services can be distinguished into message- and stream-based, a stream is in fact a single message sent in chunks or as a sequence of individual smaller messages. Languages for encoding content or media are also referred to as data serialization formats or formats in short.

Communicating parties can only parse content of a certain kind, where the format, i.e. syntax, and meaning, i.e. semantics, of the language are defined. The hardness of parsing is then a computational complexity property of the language [99]: With increasing expressiveness, more and more information can be encoded in a language but parsing also becomes harder and therefore more error-prone in software implementations [265].

Alphabets for intercommunicating digital systems are typically binary. The basic unit of information is a bit to encode two symbols. For Internet applications, a byte of eight bits is a typical transferable unit. Content can be distinguished into binary and text-based with respect to the alphabet:

- Binary content. When a language describes a bijection between digital sequences and the domain of actual values and structures, then contents are referred to as binary content and they are likely not human-readable.
- Text-based content. Text is not simply text, but rather bits and bytes with an associated mapping to human-readable symbols such that some digital sequence has a textual representation. Such a mapping is called character encoding or character set, e.g. ASCII, and content is said to be text-based, if its syntax has a human-readable representation.

ASCII is one of the simplest character encodings; it uses seven bits to enumerate a set of control and printable char-
acters but is limited to symbols popular in the English language. Unicode [295] attempts to enumerate all the human-readable symbols in all natural languages. Character encodings like the ASCII-compatible, byte-oriented UTF-8 [348] then specify how millions of Unicode symbols are digitally encoded.

2.1 Content Types

Formally, a type is a general concept shared by a set of objects, also referred to as the instances of a type. With respect to content and media, a content type is then an identifier that specifies the alphabet, character encodings, syntax, and eventually semantics of a language. The notion of content type is essential for modular software design, where an appropriate parser is chosen during runtime for some content based on its content type.

In practical applications today, the Multipurpose Internet Mail Extensions (MIME) have become an Internet standard for specifying content types in the web [82]. They are referred to as MIME content type, Internet media type, or MIME type for short. For example, the MIME type of a simple ASCII text is text/plain. A MIME type identifier can also explicitly refer to the character encoding of text-based contents [160], e.g. text/plain; charset=utf-8.

2.2 Text-Based Content

Compared to binary formats, text-based languages have a lower information density because human-readable symbols need to be digitally encoded. For example, an integer number $n > 0$ needs $\lceil \log_2 n \rceil$ bytes in human-readable UTF-8 encoding, while efficient binary representation requires only $\lceil \log_2 n \rceil$ bits. Despite lower information density, the human-readability of content has contributed the success of hypermedia in the web.

2.2.1 Semi-Structured Languages

Three influential languages for hypermedia and information exchange in the web are the Hypertext Markup Language (HTML), the Extensible Markup Language (XML), and the JavaScript Object Notation (JSON).

Hypertext Markup Language. HTML is the standard for defining websites and has MIME type text/html. It uses markup such as tags, attributes, declarations, or processing instructions to express structural, presentational, and semantic information as plain text. While the different versions of HTML up to 4.01 [303] are an application of the Standard Generalized Markup Language (SGML), which requires a complex SGML parser framework, today’s HTML5 [335] has a similar syntax, but specifies an individual parser.

An exemplary HTML5 document is listed in Figure 2. SGML-based parsers distinguish the grammars of different HTML versions in the document type declaration in the first line of a document. As HTML5 is not SGML-based, the document type declaration is deliberately incomplete to indicate SGML independence. A document is separated into a header for meta information, and a body for the actual semi-structured content of a website. All allowed tags are specified in the standard. Interestingly, the character encoding of a document is defined within the document itself in a meta tag. This tag should be the first tag in the header, so the parser becomes aware of the encoding before other tags are encountered.

An SGML or HTML5 parser in browser software transforms a document into a so-called Document Object Model (DOM) [308], a generalized tree-like data structure that is eventually rendered in a visible window. Another popular format with respect to HTML is the Cascading Style Sheets (CSS) language [322]. It allows to define the look and also behavior of a DOM’s visual representation in a window.

Extensible Markup Language. XML [319] originates from SGML, but it is more restricted and a popular format for electronic data exchange. An example is shown in Figure 3. Tags, attributes, namespaces, declarations, and processing instructions are syntactic constructs for structuring information in XML as text. The first line in an XML document should be a processing instruction that informs the parser about the XML version and the character encoding of the document.

XML is a language family: the structuring of elements (tag names) and attributes within a document is basically unrestricted, only syntactic rules from the standard have to be obliged. Element content is limited to text; by default, XML distinguishes only two datatypes, parsed (PCDATA) and unparsed character data (CDATA). Mixed-content XML relaxes element content restrictions; text in element content is also allowed to contain nested elements, e.g. the review element in Figure 3.
<?xml version="1.0" encoding="UTF-8"?>
<movie year="1968">
  <title>2001: A Space Odyssey</title>
  <director nid="nm0040">S. Kubrick</director>
  <review>A good movie.</review>
</movie>

Fig. 3 An exemplary XML document with mixed-content [144].

The underlying logical structure of XML is a tree, therefore, open- and close-tags must be correctly nested. A document with correct nesting, proper syntax, and a single root element is well-formed. Furthermore, a document is said to have an XML Information Set [312] if it is well-formed and namespace constraints are satisfied. An Infoset is an unambiguous abstraction from textual syntax, e.g. there exist two syntactic notions for empty elements in XML.

To restrict the structure of documents or Infosets, XML offers schema languages, for example, Document Type Definition (DTD) [319], XML Schema (XSD) [313], or Relax NG [183]. Formally, a schema is a form of grammar that characterizes a set of XML documents, and a document is said to be valid if its schema is obeyed [157]. In this sense, a schema allows to specify a content subtype of XML. XSD and Relax NG also support more fine-grained datatypes than PCDATA and CDATA for restricting element contents.

The duality of text representation and logical tree structure of documents has led to two different processing approaches. A document can be either parsed into a DOM for tree operations or processed directly as a stream of text, open-, or close-tag events using the Simple API for XML (SAX) [266].

XML has a certain overhead. Names of elements occur twice in open- and close-tags, and control characters like brackets are needed accordingly. SXML [140] is an alternative syntax for XML using S-expressions such that element names occur only once, and less space is required.

The MIME media type of the XML language family is application/xml. XHTML [306] is an XML-oriented specification of HTML and has a MIME type that indicates the XML origin (application/xhtml+xml). XHTML is conceptually an XML subtype with a strict syntax specified by a schema, so an XML parser can be used instead of a complex markup parser in a web browser. XML is also the supertype for many web formats, e.g. Scalable Vector Graphics (SVG) [323] or Mathematical Markup Language (MathML) [307], and MIME types for well-known subtypes are specified in [179].

JavaScript Object Notation. JSON [41] is a simple text format to serialize information as structured key-value pairs. JSON is human-readable, and an example is shown in Figure 4. The syntax is a subset of the JavaScript language (discussed in Section 2.5), and a JSON document is either parsed or evaluated to an object during runtime. JSON specifies six basic datatypes: null, Number, String, Boolean, Array, and Object, and syntactic rules to represent them as text. A proper JSON document always has a single root object.

Similar to XML, JSON defines a family of languages because there are syntactical restrictions, but no structural limitations in the standard. JSON Schema [85] is a schema language expressed in JSON format, and the motivation is the same as with XML schemas; a schema defines a set of JSON documents, and a document can be validated against a schema.

2.2.2 Binary-to-Text Encodings

Base16, Base32, Base64 [134], Percent-Encoding [33], and Quoted-Printable [81] are binary-to-text encoding schemes to map an arbitrary binary value to a text of ASCII printable characters. Most notably, Base64 encoding is a popular method for embedding arbitrary binary content in XML or JSON as text, but incurs a 33% overhead due to reduced information density.

2.2.3 Other Text-Based Formats

There exist several specifications for text-based languages with varying popularity, for example: comma-separated values (CSV) [270] for relational data; markup languages Candle [44] and YAML [30]; the Ordered Graph Data Language (OGDL) [296] for graph-structured data; and the Open Data Description Language (OpenDDL) [149] for structured information.

2.3 Binary Content

Binary formats offer compact representation of information. Contents are typically not human-readable, and examples include audio and video formats.

Abstract Syntax Notation One (ASN.1) [62] is a popular standard for structured information exchange. ASN.1 distinguishes between an abstract specification of information
structure and encoding rules such as Basic Encoding Rules (BER) or Distinguished Encoding Rules (DER). An encoding rule defines how an actual instance, according to some abstract structure, translates to bits and bytes. An application of ASN.1 using DER are X.509 certificates in today’s public key infrastructures [250].

As text-based languages introduce overhead, several binary equivalents to popular text-based formats have been proposed. With respect to binary representations of XML, there are standardized attempts: Efficient XML Interchange (EXI) [334]; .NET Binary XML [168]; Fast Infoset [127] as an application of ASN.1; and Binary MPEG Format for XML (BiM) [125] which originates from a video format.

There are also attempts to optimize XML representation through binary equivalents such as: Binary JSON [43]; MessagePack [84]; and Concise Binary Object Representation (CBOR) [37]. Figure 5 gives an example, how MessagePack translates the JSON document from Figure 4 into a more compact binary form.

Several discussed Remote Procedure Call (RPC) architectures (Section 5.2.1) specify individual binary formats to serialize structured data, e.g.: External Data Representation (XDR) [171], Apache Avro [13], Apache Etch [11], Apache Thrift [10], Protocol Buffers [89], and Hessian [47].

Other binary languages worth mentioning are the Structured Data Exchange Format (SDXF) [342] for hierarchically structured information and Property List [18], a serialization format in Apple systems.

2.4 Container Formats

A container is a special form of encoding for nesting other arbitrary contents. The MIME standards also specify a multipart [82] content type for containers, where contents of varying types are interleaved. The core idea in multipart is a text-based boundary string that separates individual parts as shown in Figure 6. Each part has an individual header that denotes its Content-Type and additional metadata such as binary-to-text encodings. Multipart defines several subtypes for different applications:

- multipart/alternative [82] to model a choice over several contents to a consumer;
- multipart/byteranges [68] to characterize a subsequence of bytes that belongs to a larger message or file;
- multipart/digest [82] to store a sequence of text-based messages in a single content;
- multipart/form-data [158] for submitting a set of completed form fields from an HTML website;
- multipart/mixed [82] for an email message;
- multipart/mixed for inline placement of media in a text-based message, e.g. images in emails;
- multipart/parallel [82] to display all parts simultaneously on hardware or software capable of doing so;
- multipart/related [151] as a mechanism to aggregate related objects in a single content;
- multipart/report [143] as a container for email messages; and
- multipart/x-mixed-replace [181] for a list of parts that are sequentially presented, and the most recent part always invalidates all preceding ones, e.g. for streaming of events.

An application of multipart is XML-binary Optimized Packaging (XOP) [320]. XOP specifies a container format for XML as MIME multipart/related package [151], where binary element content is natively embedded to maintain information density without the necessity of binary-to-text encoding. The W3C furthermore specifies a set of attributes and rules for XML and XSD to describe media content as MIME types [314] which is used in XOP.

S/MIME [247] is a security extension for MIME. It defines encryption (multipart/encrypted) and digital signatures (multipart/signed) for confidentiality, integrity, and non-repudiation of data using public key cryptography. A drawback of MIME multipart is that the boundary string must not appear in any of the parts because it would break the format. Microsoft’s Direct Internet Message Encapsulation (DIME) [165] is a standard for encapsulation and streaming of arbitrary binary data in the spirit of MIME.
multipart, but with an improved boundary mechanism to distinguish parts.

2.5 JavaScript

ECMAScript [64], better known as JavaScript, is a dynamic programming language which is supported by practically all available web browsers today. JavaScript can be distinguished into text-based syntax and a JavaScript runtime environment. A runtime environment is typically present in web browser software but also available for service implementation, e.g. the Node.js [135] framework, or apps.

JavaScript code is either embedded directly into HTML or XHTML markup using script tags, inlined in attributes, or exchanged as text-based content. When exchanged as resource, JavaScript code has an individual MIME type, i.e. application/javascript [105]. A browser runtime environment then interprets the code to dynamically adapt the DOM and its visual representation. While JavaScript is in fact a Turing-complete programming language, its runtime environment enforces restrictions on system and network access during execution, e.g. local file access, to implement a security policy.

3 Service Interaction Patterns

Clients and services exchange messages of certain content types for interaction. The service interaction patterns introduced by Barros, Dumas, and Ter Hofstede [24,25] are recalled in Figure 7 for completeness. The patterns characterize styles of message exchange between parties in a distributed network, and the authors propose three dimensions to distinguish patterns: the number of participating parties; the number of message transmissions in an interaction; and whether messages in two-way interaction are routed to third-parties or take round trips. This leads to four pattern groups: single-transmission bilateral, single-transmission multilateral, multi-transmission, and routing patterns.

This section is a high-level summary of the descriptions in [25]. The patterns provide unified terms to discuss protocols and architectures in later sections.

3.1 Single-Transmission Bilateral Patterns

Send. Party A sends party B a message.
Related to: unicast, point-to-point send.

Receive. Party A awaits an incoming message.
Related to: listener, event handler.

Send-Receive. Party A sends party B a message and causes B to respond with a message. This pattern has a dual, i.e. Receive-Send, where party A waits for an incoming message and returns a response when received.
Related to: request-response, request-reply, RPC.

3.2 Single-Transmission Multilateral Patterns

Racing Incoming Messages. Party A waits for a single incoming message from a number of possible messages and senders. The continuation of A after receiving the first message depends on the message type or sender, and subsequent messages may or may not be discarded.
Related to: racing messages, deferred choice.

One-to-Many Send. Party A sends n messages to the parties \( B_1 \ldots B_n \), where all messages have the same type but not necessarily the same content.
Related to: multicast, broadcast (where a third-party acts as broker to address all parties in the domain), event notification, scatter, publish-subscribe, fan-out.

One-from-Many Receive. Party A awaits a number of logically connected messages from senders \( B_1 \ldots B_n \). The messages have to arrive within a certain time span so they can be linked together.
Related to: event aggregation, gather, fan-in.

One-to-Many Send-Receive. Party A sends n request messages to recipients \( B_1 \ldots B_n \) and waits for a certain time span for responses from the recipients. Whether A considers the interaction successful or not depends on the response messages that arrive in time. This pattern has also a dual, i.e. One-from-Many Receive-Send, where party A waits for a certain time for messages from \( B_1 \ldots B_n \), then processes them and returns individual responses.
Related to: scatter-gather.

3.3 Multi-Transmission Patterns

Multi-Responses. Party A sends a request to party B. B then returns responses until some stop-condition is met. Possible stop-conditions are an explicit notification from A, a deadline given by A, an interval of inactivity, or an explicit notification from B that the stream has stopped.
Related to: streamed responses, message stream.

Contingent Requests. Party A sends a request message to \( B_i \). If there is no response within a certain time, A sends the request to \( B_2 \), and if there is again a timeout, A continues this cycle until some \( B_i \) responds properly.
Related to: send with failover.

Atomic Multicast Notification. Party A sends notifications to parties \( B_1 \ldots B_n \), where a minimum number of i and a maximum number of j recipients are required to accept the notification. A constraint of \( i = j = n \) means that all recipients have to accept.
Related to: transactional notification.
3.4 Routing Patterns

Request with Referral. Party $A$ sends a request to party $B$, where the message is evaluated, and based on certain conditions, e.g. message content, follow-up responses are forwarded to a single or multiple parties $C_1 \ldots C_n$. Related to: reply-to.

Relayed Request. Party $A$ sends a request to party $B$, and $B$ relays it to parties $C_1 \ldots C_n$, who take on further interaction with $A$. Party $B$ still retains a view of the ongoing interaction between $A$ and $C_1 \ldots C_n$. Related to: delegation.

Dynamic Routing. There exist routing conditions that define how a message from party $A$ is forwarded. A routing condition can be dynamic, i.e. depend on message content. Based on the conditions, a request from $A$ is forwarded to one or more parties $B_1 \ldots B_n$ that process and eventually forward the message to $C_1 \ldots C_n$. These parties then again process, continue to apply routing conditions and so on. Related to: routing slip, content-based routing.

4 Protocols

To deal with the inherent complexity of communication in computer networks, layered protocol design, as proposed in the conceptual OSI reference model [353] or the simplified Internet model [40], has become an industry standard to separate concerns in protocol development.

Low delay and multiplexing are two major drivers for recent developments in accelerating web technology. Multiplexing in this context refers to techniques for transporting multiple parallel dialogues over a single channel between two peers. Network protocols are therefore recalled, and the survey approaches the state-of-the-art for bilateral and multilateral information exchange in a bottom-up fashion. Figure 8 shows the Internet model and a number of communication protocols popular for the web and services. Protocols in the link layer specify how devices with physical interfaces and physical addresses can exchange information when they are connected. Examples include IEEE 802.3 Ethernet or IEEE 802.11 Wi-Fi networks.

The Internet layer allows hosts to communicate beyond their local neighborhood of physically connected devices. Logical addresses, routing, and packet-based data exchange are the core aspects of this layer.

The transport layer enables inter-process communication over networks. Processes can run on the same host and share the same logical network address. Transport layer protocols extend the logical addressing and enable communication between distributed processes.

Application layer protocols finally dictate how to provide functionality, content and media across two or more processes that are able to communicate, e.g. clients and services. These protocols enable the web and SOA, and they characterize communication between processes.

While Internet and transport layer protocols are typically provided by operating systems, application layer protocols are implemented by the client and service. From a lexical point of view, all Internet protocols share the same binary alphabet. A byte is considered as the smallest transferable symbol in link layer technologies such as Ethernet or Wi-Fi. A common practice seen in Internet protocols is to separate a transferable sequence of bytes into a protocol header and content, where the header specifies the content’s type, so other protocols can be recursively embedded. Due to the
Technologies for Web and Cloud Service Interaction: A Survey

The TCP/IP protocol suite [276] is the de facto standard for computer networks. Internet layer functionality is provided by the Internet Protocol (IP) [238], or IPv4, which defines packet-based networking by an addressing schema, packet layouts, and routing conditions. Using IP, host A can send a packet to the logical address of host B without knowing the physical address or location of B. Based on the addresses in the header of the packet, so-called routers forward the packet, but delivery is not guaranteed. IP therefore implements a Dynamic Routing pattern. To send and receive packets, a host needs at least one logical IP address in a physically connected network, and a router (gateway) in this network that forwards packets from and to the host. If a host has simultaneous links to two or more physical networks, it is referred to as multihoming.

The maximum size of an IP packet is bounded by the underlying link layer technology, and packets are eventually fragmented or dropped if they are too large for a link on the network path. Fragmentation allows to split oversized packets into smaller ones, but increases the load on a link because more packets result in larger overhead. The header of an IP packet also specifies the type of the enclosed transport protocol in the content.

IP has restrictions; the upper bound of $2^{32}$ logical addresses is the most serious issue given the growth of the Internet. An ad-hoc solution is Network Address Translation (NAT) by routers for private networks. There is an ongoing effort to switch to IP Version 6 (IPv6) [59] that provides $2^{128}$ logical addresses to deal with the exhaustion problem that becomes imminent in an Internet of Things.

For addressing multiple recipients with a single packet, IP offers so-called broadcast addresses for a group of recipients based on the logical addressing scheme, i.e. a subnet. IPv6 does not support broadcasts. Both IP and IPv6 offer multicast, where a packet, sent to a special logical group address, is replicated for all recipients in the group. Both multicast and broadcast enable the One-to-Many Send pattern on top of Dynamic Routing.

Ideally, the Internet layer promotes content neutrality. All routing decisions should depend on the header of IP packets independent from their content. But this neutrality does not hold in practice. Actual networking devices like firewalls, security filters, Quality-of-Service traffic shapers or content-based routers derive routing decisions from the contents of IP packets. Such devices have functionality that extends across layers in reference models. Therefore, they are referred to as middleboxes [45], and their preferred techniques are grouped under the term Deep Packet Inspection (DPI) [31].

### 4.1 Internet Layer Protocols

The TCP/IP protocol suite [276] is the de facto standard for computer networks. Internet layer functionality is provided by the Internet Protocol (IP) [238], or IPv4, which defines packet-based networking by an addressing schema, packet layouts, and routing conditions. Using IP, host A can send a packet to the logical address of host B without knowing the physical address or location of B. Based on the addresses in the header of the packet, so-called routers forward the packet, but delivery is not guaranteed. IP therefore implements a Dynamic Routing pattern. To send and receive packets, a host needs at least one logical IP address in a physically connected network, and a router (gateway) in this network that forwards packets from and to the host. If a host has simultaneous links to two or more physical networks, it is referred to as multihoming.

The maximum size of an IP packet is bounded by the underlying link layer technology, and packets are eventually fragmented or dropped if they are too large for a link on the network path. Fragmentation allows to split oversized packets into smaller ones, but increases the load on a link because more packets result in larger overhead. The header of an IP packet also specifies the type of the enclosed transport protocol in the content.

IP has restrictions; the upper bound of $2^{32}$ logical addresses is the most serious issue given the growth of the Internet. An ad-hoc solution is Network Address Translation (NAT) by routers for private networks. There is an ongoing effort to switch to IP Version 6 (IPv6) [59] that provides $2^{128}$ logical addresses to deal with the exhaustion problem that becomes imminent in an Internet of Things.

For addressing multiple recipients with a single packet, IP offers so-called broadcast addresses for a group of recipients based on the logical addressing scheme, i.e. a subnet. IPv6 does not support broadcasts. Both IP and IPv6 offer multicast, where a packet, sent to a special logical group address, is replicated for all recipients in the group. Both multicast and broadcast enable the One-to-Many Send pattern on top of Dynamic Routing.

Ideally, the Internet layer promotes content neutrality. All routing decisions should depend on the header of IP packets independent from their content. But this neutrality does not hold in practice. Actual networking devices like firewalls, security filters, Quality-of-Service traffic shapers or content-based routers derive routing decisions from the contents of IP packets. Such devices have functionality that extends across layers in reference models. Therefore, they are referred to as middleboxes [45], and their preferred techniques are grouped under the term Deep Packet Inspection (DPI) [31].

### 4.2 Transport Layer Protocols

Transport layer protocols enable inter-process communication over networks. Protocols can be distinguished by certain characteristics like:

- Uni- or bidirectional communication;
- Connection-oriented (stateful) or stateless;
- Message- or byte-stream-based information exchange;
- Ordering of messages or bytes in the stream;
- Reliable delivery;
- Data integrity;
- Flow and link congestion control.

The more properties a protocol supports, the more overhead for control structures is required, and timeliness is affected. This leads to an upper bound for the maximum transfer rate. Nonetheless, there is always a time delay between sending and receiving information. If, for example, a protocol requires several interactions to synchronize state or acknowledgments for reliable delivery, the delays accumulate and effectively limit the available transfer rate.

The two most prominent transport layer protocols in the Internet are the Transmission Control Protocol (TCP) [239] and User Datagram Protocol (UDP) [237]. Both protocols are available in modern operating systems. There is an increased interest in enhanced protocols, such as MultiPath TCP (MPTCP) [78], Stream Control Transmission Protocol (SCTP) [277], and Google’s Quick UDP Internet Connections (QUIC) [254] to overcome limitations of TCP and UDP.
**Transmission Control Protocol.** TCP is a protocol to connect two endpoints, e.g., processes, and information is exchanged in bidirectional byte streams, where the correctness and order of bytes in both streams are guaranteed. To be compatible with packet-based IP delivery, a continuous stream is split into so-called segments. A TCP connection is stateful and establishes a session between the two endpoints. It requires a so-called three-way handshake to synchronize which causes a delay before the streams can start. TCP distinguishes a client that initiates the handshake, and a server that listens for incoming connections. An attempt to reduce the latency caused by the handshake between two already familiar endpoints is the TCP Fast Open extension [49]. A source and destination port number in a TCP segment header addresses the client and service processes or endpoints on a host. There is no identifier in TCP that refers to the transported application layer protocol – TCP just transmits bytes. IANA therefore maintains a list of default server ports that are also assumed by URI schemes if not explicitly overridden, e.g., port 80/TCP for HTTP.

Delivery in TCP is reliable through acknowledgments and retransmissions. Integrity is guaranteed by checksums. TCP segment headers also contain a window size that informs the receiver how many bytes the sender can receive in the other directional stream. The window mechanism enables flow and congestion control through rate adaptation. The maximum segment size is announced as a TCP option to avoid IP fragmentation. A problem in TCP is so-called head-of-line blocking; if a single byte in a stream is incorrect or lost on the path, the stream cannot proceed until the missing part is correctly redelivered. This incurs a problem for messaging protocols implemented on top of TCP. The necessity of acknowledgments limits TCP to bilateral communication; in other terms, TCP provides Send and Receive.

**User Datagram Protocol.** UDP is a message-based unidirectional protocol with a minimal header. UDP transports datagrams without acknowledgments and in any order. A datagram header only holds a source and destination port, the length of the content, and a checksum for integrity. There is also no support for flow or congestion control. The overhead of UDP is small and it is stateless, therefore no synchronization is required beforehand. If the size of a datagram exceeds the maximal content length of the IP packet, fragmentation takes place. Similar to TCP, a datagram is a sequence of bytes, and there is no application layer protocol identifier in UDP. IANA also assigns default server ports to UDP-based protocols.

UDP supports the Send and Receive pattern without any guarantees of delivery. Because it is stateless, UDP is also a candidate for multilateral communication; together with IP broadcast or multicast, where datagrams are replicated by the networking infrastructure, One-to-Many Send is possible.

**MultiPath TCP.** A drawback of TCP is that segments in an established connection eventually take the same network path, and a network problem disrupts communication until error handling routines or timeouts are triggered. This problem affects mobile devices in radio dead spots or during roaming between wireless networks. Especially for mobile devices, it is more and more common that a device is connected to several physical networks simultaneously, i.e., multihoming. MPTCP is an extension for TCP to increase both redundancy and transfer rate by aggregating multiple paths over all available links as subflows of a single connection [35]. The MPTCP connection does not fail if a path becomes congested or interrupted and an alternative path is still available. Semantically, an MPTCP connection behaves like a TCP connection.

For compatibility with middleboxes, MPTCP is a host-side extension, and subflows are regular TCP connections. A notable example using MTP is Apple Siri which utilizes both Wi-Fi and 3G/4G networks for increased service availability in mobile devices [19]. In terms of patterns, communication in MPTCP is still bilateral like TCP, i.e., Send and Receive.

**Stream Control Transmission Protocol.** TCP offers reliable delivery and strict order in the stream, but there are applications that require reliability and ordering is less important; TCP can create unnecessary delays in such a scenario [277]. Also, the streaming nature of TCP introduces complexity in higher protocols for messaging because streams then need a notion of state, delimiters to indicate message boundaries, and specific measures to circumvent head-of-line blocking. SCTP [277] is an alternative to TCP that was designed to raise availability through multihoming as seen in MPTCP. An association between two endpoints is established in a four-way handshake. A handshake needs more interactions than in TCP but the SCTP service endpoint stays stateless until synchronization is completed. This eliminates the well-known security vulnerability of TCP SYN flooding [65].

SCTP is message-based and multiplexes byte streams of messages, similar to MPTCP subflows, into a single association. Also, SCTP offers reliability through checksums, optional ordering, and rate adaptation for flow and congestion control, similar to TCP [210]. Messages are assumed to be sequences of bytes, and SCTP does not identify the actual application layer protocol of a message. Similar to TCP and UDP, default server ports are managed by IANA. A drawback of SCTP is its limited popularity. Transportation over the Internet is therefore not guaranteed because middleboxes might not be aware and block it. The supported interaction patterns are Send and Receive.
Quick UDP Internet Connections. Developed by Google and already available in the Chrome browser, QUIC [254] is an experimental transport layer protocol to reduce latency and redundant data transmissions in the web. QUIC implements the multiplexing concept from SCTP on top of UDP to overcome the issues of SCTP communication being filtered by middleboxes. Latency is reduced because no handshake is required between familiar hosts. QUIC also supports data compression, provides checksums and retransmission for reliability, and supports congestion avoidance. Forward error correction minimizes obstructive retransmissions by an error-correcting code. In terms of patterns, QUIC offers bilateral Send and Receive.

Other Transport Protocols. There exist several transport layer protocols, where no evident application in a web or cloud context has been found. Examples for multilateral interaction are multicast transport protocols [208]. Examples for for bilateral interaction are: the message-based Datagram Congestion Control Protocol (DCCP) [142] that offers congestion control but its delivery is unreliable; UDP Lite [148] with relaxed checksum calculation; and Reliable UDP (RUDP) [39] as an extension of UDP with acknowledgements, retransmissions, and flow control.

4.3 Transport Security

Several standards for establishing secure sessions, independent from applications, have emerged to achieve confidentiality, integrity, and authenticity in a communication session. This “virtual” security layer is situated between transport and application layer in the Internet model or in the OSI model’s session layer.

The most prominent protocols for encryption are the Secure Sockets Layer (SSL) [83] and Transport Layer Security (TLS) [60]. Both operate on top of TCP and establish an encrypted session for byte-stream-based information exchange. Historically, SSL was invented by Netscape and the latest version 3.0 became obsolete with TLS 1.0 in 1999. Today’s TLS 1.2 still offers limited backward compatibility to SSL.

To establish an encrypted session in SSL/TLS, an initialization routine, i.e. SSL or TLS handshake, is necessary. The most notable difference between SSL and TLS is when the handshake happens. An SSL handshake takes place implicitly after the TCP handshake, independent from the application. But TLS explicitly allows to trigger a handshake from the actual application by issuing the STARTTLS command to upgrade an already established TCP connection. During the handshake, the communicating parties authenticate themselves through X.509 public key infrastructure and certificates, they agree on a cipher suite for the session, and securely exchange session keys. The handshake requires several interactions and adds a delay before encrypted communication can proceed. For more details the author refers the reader to Rescova’s book [250].

SSL/TLS only works with TCP because the data transfer is assumed to be correct and in order – A TLS handshake or session will fail otherwise. In MPTCP, subflows are regular TCP connections, and TLS is therefore supported [267]. But TLS is challenging for transport protocols using multiplexing such as SCTP and QUIC. If a single byte is lost or corrupt, all multiplexed streams will be stalled. SCTP also supports unordered delivery of messages and partial reliability, which conflicts with the reliability and order assumptions of TLS [278, 290]. One possibility is to establish a TLS session for every individual stream in the multiplexed connection, but the numerous TLS handshakes accumulate delays and affect timeliness. TLS is therefore not an optimal choice to secure SCTP and QUIC.

Datagram TLS (DTLS) [251] is a modification of TLS that operates on top of UDP. It specifies a record type as message container, sequence numbers to detect out-of-order records, and retransmission conditions. DTLS is also standardized for other transport protocols, for example, DTLS over DCCP [234] and DTLS for SCTP [290]. QUIC specifies its own cryptography protocol loosely based on TLS, and all transmissions are encrypted by default to avoid tampering by middleboxes [147].

4.4 Application Layer Protocols

Protocols that define inter-process communication on top of transport protocols are referred to as application layer protocols. These protocols are often simple service protocols implementing the Send-Receive pattern, where a server waits for incoming communication. They can be informally distinguished by properties:

- Transport layer assumptions;
- Text-based or binary protocol;
- Stateful or stateless;
- Combined or separated data and control connections.

Transport protocols like TCP and UDP have different properties that affect inter-process communication. An application layer protocol assumes properties of transport and therefore restricts the allowed transport layer protocols. Another distinction for application layer protocols is whether their syntax is human-readable. Text-based protocols typically have an overhead because of less efficient information encoding. A protocol is said to be stateful if the result of a previous interaction affects the choice of the next interaction. Another distinction for protocols that transfer other media is whether the protocol and media syntax are merged into a single stream or message.
It should be noted that some protocols distinguish between a conceptual high-level syntax and a wire format that is actually transmitted. For example, while high-level syntax could be text-based, the effectively sent data could be compressed for efficient transportation.

Two well-known text-based application layer protocols are the File Transfer Protocol (FTP) [240] and the Simple Mail Transfer Protocol (SMTP) [141]. Both require TCP and support TLS for security; while FTP maintains two separate TCP connections for text-based control commands and binary media transfer, SMTP sends the text-based control commands and the email message in the same TCP connection. Both protocols are stateful because they require several Send-Receive interactions, where success or failure of the current interaction decides the next action.

4.4.1 Domain Name System

The Domain Name System (DNS) [174, 175] is an Internet core service, managed by IANA, and specifies an application layer protocol. A distributed database for a hierarchical naming scheme maps names onto IP addresses because names are more usable for humans than numeric addresses. Records in DNS have a certain type, e.g. type A is a host address record, type CNAME is an alias for another name, or type MX is reserved for SMTP-service-specific records. The hierarchical name of a host is then referred to as Fully-qualified Domain Name (FQDN).

DNS is a binary and stateless protocol implementing the Send-Receive pattern. The client queries the service to resolve a name of a certain type, and the service eventually returns a record. DNS uses UDP as primary transport protocol for queries but also supports TCP for large responses or DNS transactions, i.e. zone transfers. The drawback of using TCP for short queries is the handshake delay.

In fact, DNS has become a critical service for today’s Internet. Basically all web technologies rely on DNS as an abstraction layer for locating service endpoints; DNS service records (type SRV) even enable dynamic service discovery [50]. Nevertheless, a failure, misuse, or misconfiguration in DNS can lead to unforeseeable security consequences, e.g. attacks like DNS spoofing [100] are a serious threat. If DNS responses are tampered with, an attacker can redirect interaction to malign hosts. DNSSEC [20, 21, 22] is a recent attempt to secure correctness and authenticity of queries and responses using cryptographic methods.

4.4.2 Resource Identification and Location

A so-called Uniform Resource Identifier (URI) is a human-readable text string that identifies a resource in the web. The URI specification [33] defines the syntax of a URI and also describes a Uniform Resource Locator (URL). A URL is a subtype of URI that both identifies and locates a resource.

Figure 9 shows an example. The URI scheme identifies the application layer protocol for accessing the resource and implicitly defines the TCP or UDP port of the service. Schemes and default ports are standardized for application layer protocols by IANA, e.g. for FTP (ftp:) or SMTP (mailto:). The authority contains a Fully-qualified Host Name (FQHN) which is either a FQDN or an IP address to locate the host of the service. Also, additional user information and alternating TCP or UDP port information can refine the authority. The path identifies a resource in the authority, an optional query part holds a list of key-value pairs, and an optional fragment can link to specific information within the resource.

A URL has a notion of origin. Informally, two URLs share the same origin if they have identical scheme, FQHN, and port [27]. As URLs play a central role in the web, there is a trend to keep URLs as simple as possible to increase usability. So-called clean URLs [213] have an expressive path and avoid the query part, e.g. the clean URL for Figure 9 is “http://www.ex.com/fotos/view#page2”. The rewriting of URLs into a clean form, automatically and manually, is a common practice in web development.

4.4.3 The Hypertext Transfer Protocol

HTTP, originally specified in [68], is the most fundamental application layer protocol for the web and many service technologies. It is stateless and implements the Send-Receive pattern: a client sends a request message, and the service answers with a response message. HTTP is designed to operate above TCP, both control and content are sent in the same TCP connection, where control instructions are text-based. To separate control from data, HTTP specifies a header format and delimiters. Figure 10 shows an example Send-Receive cycle between a client and a service.

Both HTTP request and response messages specify a header and an optional body separated by a delimiter. The request header characterizes the method, the URI-path of a resource, the protocol version and a list of header fields. The presence of a body depends on the method. Similarly, the response header holds a status code, a list of response header fields and eventually a body, i.e. the requested content.

Header fields make HTTP expandable – Some header fields are mandatory, others are optional. A central header
After the client has established a TCP connection with the service, a request is sent as text-based stream. The service parses the request and returns a response containing a header and the resource through the other directional stream of the TCP connection. Depending on the HTTP method, a request eventually has a content, e.g. from a form submission or file upload.

Field is Content-Type to specify the MIME type in a message carrying content. The content type of a requested resource is therefore undefined until the HTTP response message arrives at the client.

Today’s state of the art version HTTP/1.1 optimizes its predecessors in several ways. HTTP/1.1 supports a total of eight methods: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, and CONNECT. The Upgrade header field enables the client and service to completely switch the application protocol after a Send-Receive cycle. The Host header allows to distinguish different FQDN authorities that are hosted on the same IP address – a common practice in web hosting.

Also, HTTP/1.1 introduces persistent TCP connections and pipelining of requests to minimize the delay caused by numerous TCP handshakes. The standard proclaims that a client should not exceed two simultaneous TCP connections to a service. HTTP differs explicit Content-Encoding of the requested resource and Transfer-Encoding for implicit compression between client and service or a so-called proxy. HTTP also offers fine-grained caching mechanisms, e.g. by timestamps, the ETag header, and conditional GET, such that clients and proxies can minimize data transfer.

HTTP/1.1 allows client- or service-driven content negotiation for a resource [74]. For service-driven negotiation, the service respects the client’s User-Agent for sending a suitable representation, but also considers client-side content type restrictions (Accept), accepted character encodings for text-based formats (Accept-Charset), accepted content encodings (Accept-Encoding), and personal natural language preferences (Accept-Language). In client-driven negotiation, also referred to as agent-driven, the service returns a multiple-choices status in a first response that lists all available representations; the client can then choose in a second Send-Receive cycle.

While HTTP is stateless in principle, HTTP/1.1 adds support for so-called cookies using the Set-Cookie and Cookie header fields to track state across Send-Receive cycles for a session [26]. A cookie is basically a text string that identifies a client’s HTTP session. The responsibility for correct application state tracking is on the service-side; cookies therefore have important security and privacy aspects.

Recently, the HTTP/1.1 standard has been completely respecified in order to remove imprecisions that have led to ambiguous implementations [69,70,71,72,73,74].

HTTP Security. The use of SSL/TLS for TCP connections has become the de facto standard to secure HTTP interaction between a client and a service, and it is specified as HTTP Secure (HTTPS) [249]. As HTTPS has its own URI scheme (https://), a user can recognize from a URL whether accessing a resource is protected.

A client can eventually choose to access an identical resource through multiple transport mechanisms, e.g. HTTP or HTTPS. To notify a web client that resources of a certain authority can only be accessed through HTTPS, the HTTP Strict Transport Security (HSTS) [104] specifies a response header field that informs the client about this policy.

Using the CONNECT HTTP method, a client can ask a proxy service to establish a connection to the intended ser-
vice on behalf of the client, and interacted byte streams are forwarded. This functionality is referred to as HTTP tunnelling and necessary in proxies to enable access to HTTPS services. In this case, no caching is possible because data is encrypted.

Another way to secure HTTP interaction is TLS to upgrade an existing TCP connection to an encrypted one using the Upgrade header field in HTTP/1.1 [137]. A drawback of this solution is that a user can no longer see from a URL whether access is encrypted or not.

**Push Technology.** In terms of patterns, a Send-Receive interaction in HTTP is synchronous and can only be initiated by the client; resources are pulled from a service. Due to the wide availability of HTTP-enabled software and wide acceptance by middleboxes, HTTP has also been exploited to achieve asynchronous interaction without breaking the protocol specification, e.g. for client-side Receive or Multi-Responses patterns. These techniques are referred to as push technology [3] such that a service can push a resource to a client preferably in real-time, e.g. for data feeds or event notification, without being explicitly requested.

Historically, the first attempts have resorted to polling on the client-side, and real-time event notification was not possible. To minimize the number of Send-Receive cycles and to decrease response times, long polling is similar to polling, but the HTTP request hangs until a server-side event or a timeout occurs.

Comet [257], also known as HTTP Streaming or HTTP server push, exploits persistent connections in HTTP/1.1 to keep a single TCP connection open after a client requests an event resource. The service then gradually delivers events using MIME type multipart/x-mixed-replace for the response. Comet implements the Multi-Responses pattern.

Reverse HTTP [150,86,271] exploits the Upgrade feature of HTTP/1.1 to change the application layer protocol and switch the roles of client and service. The service becomes a client with respect to the established TCP connection, and real-time events are then issued in form of HTTP request messages from the original service to the original client, i.e. client-side Receive-Send terms of patterns.

For simultaneous bidirectional communication between client and service, Bidirectional-streams Over Synchronous HTTP (BOSH) [226] maintains two separate TCP connections. The client uses the first connection to issue HTTP request messages to the service, the second connection is a hanging request initiated by the client, so the service can interact with the client asynchronously. This enables patterns Send and Receive for both client and service.

Two recent web techniques in HTML5 are Server-sent Events (SSE) [327] and WebSocket [67,328]. For SSE, the client requests a resource that acts as event resource similar to Comet, i.e. implements the Multi-Responses pattern. The response is of MIME type text/event-stream, the TCP connection is kept open, and events are delivered as byte chunks. In case of a timeout, the client reconnects to the event resource.

WebSocket establishes a bidirectional channel for simultaneous communication in both directions. A WebSocket connection behaves like a bidirectional byte-stream-oriented TCP connection between client and service for Send and Receive patterns of interaction. A connection is established in a handshake by exploiting the Upgrade feature of HTTP/1.1. During this Send-Receive cycle for the handshake, WebSocket properties are negotiated in form of HTTP headers, including Sec-WebSocket-Protocol to agree on a sub-protocol for continuation after the handshake. WebSocket supports operation on top of TLS and has individual URI schemes for unencrypted (ws:) and encrypted (wss:) communication.

With respect to Comet, Reverse HTTP or BOSH, WebSocket has the smallest overhead because it is independent of HTTP when established. Nevertheless, it needs to agree on an explicit subprotocol on both client- and service-side for further operation.

**Performance and Speed.** Performance is an issue in HTTP when many resources are requested simultaneously from a service. Even when HTTP/1.1 persistent connections and pipelining are supported, access becomes somehow serialized because of limited simultaneous TCP connections. Allowing more parallel connections has a negative effect on the availability of a service because there is an upper limit in TCP, how many connections a host can serve simultaneously. A potential workaround for the connection limit is domain sharding [273]; resources are distributed over multiple authorities, controlled by the provider, so a client can use the maximum number of parallel TCP connections to every authority.

In general, there are a several approaches to increase web performance and user experience:

- minimize protocol handshake latency;
- reduce protocol overhead;
- multiplexed access;
- prioritization of access.

Two standards that have never left the experimental stage are the specification of the binary HTTP-NG [301], intended as a successor of HTTP, and multiplexing HTTP access over a single TCP connection based on SMUX [302]. Other experimental approaches for multiplexing are Structured Stream Transport [79] and HTTP over SCTP [180].

The state of the art, Google SPDY [284], acts conceptually as a session layer protocol between TCP and HTTP to increase web performance through multiplexed resource access, prioritization and compression of headers, and content
to reduce overhead. SPDY changes the wire format but retains semantics of HTTP; it is basically an augmentation to HTTP and no individual URI scheme is specified for compatibility reasons. SPDY also allows service-initiated interaction to push related resources to the client before they are asked for, i.e. the Multi-Responses interaction pattern.

As SPDY changes the wire format of HTTP, encryption is mandatory to prevent middleboxes from tampering with interactions. SPDY requires TLS with Next Protocol Negotiation (NPN) support for backward compatibility with the HTTPS. When a client accesses an HTTPS service that supports SPDY, the service announces SPDY through NPN during the TLS handshake, and the client can choose to proceed either with SPDY or traditional HTTP within the TLS session. The experimental QUIC transport protocol was specifically designed for SPDY to remove delays between familiar hosts, caused by the initial TCP handshake, and to optimize flow control [254].

SPDY and WebSocket have been proposed as two core technologies for the upcoming HTTP/2 which is currently in the specification process [224]. SPDY is already confirmed as the basis for HTTP/2, but protocol negotiation will be switched from NPN to the more general TLS Application Layer Protocol Negotiation (APLN) mechanism in the future [29]. Contrary to SPDY, HTTP/2 will also support unencrypted access to resources.

### 4.4.4 Web Client-to-Client Communication

A recent development in web clients, already supported by several modern browsers, is Web Real-Time Communications (WebRTC) [95,332] to enable direct interaction between clients. The motivation is real-time information exchange without the necessity of a third-party service or broker for video and voice calls to avoid network delays and bottlenecks.

WebRTC uses UDP for transmissions and still needs a service for initial discovery and signaling between clients, but also for dealing with NAT or firewalls on the path between the two clients. But WebRTC is not limited to audio and video streaming. For general information exchange, WebRTC features a so-called RTCDataChannel to establish a direct SCTP association, protected by DTLS, between two clients [253]. Interaction is therefore message-based and a client can choose whether the association provides ordering or reliable transfer. An RTCDataChannel supports Send and Receive interaction patterns between clients and will therefore blur the distinction between clients and services in the future.

### 4.4.5 Messaging Protocols

An increasingly popular group of transport protocols is for messaging in distributed systems, where participating peers interact through a message-oriented middleware or a message broker service. Messaging solutions often specify their own wire formats and this subsection enumerates the most important ones with respect to services applicable in cloud computing. Concrete architectures utilizing these protocols and their characteristics are then discussed in Section 5.2.4.

#### Individual Wire Formats

The Microsoft Message Queuing (MSMQ) [167] service specifies individual wire formats and utilizes various transport protocols, i.e. TCP and UDP, during interaction. Version 3.0 of MSMQ also introduces messaging through HTTP or HTTPS to overcome middleboxes. For sending messages to multiple recipients on different hosts, MSMQ supports IP multicast, so message replication is performed by the network, and not MSMQ.

TIBCO offers several proprietary messaging solutions such as the Enterprise Message Service (EMS) [287] or Rendezvous [288] for enterprise and cloud messaging services. While EMS utilizes individual XML-based message formats sent over TCP or WebSocket [289], Rendezvous uses a proprietary binary message format sent over UDP and IP multicast for One-to-Many Send interaction.

The OpenMQ binary wire format is a proprietary protocol for message queues in the Java software implementation Glassfish [88]. Another individual binary format is OpenWire [7] for a message queue software implementation named Apache ActiveMQ [6]. Apache Kafka [17] also specifies an individual binary wire format on top of TCP transportation. ZeroMQ [115] is an intelligent socket library for exchanging arbitrary binary messages that are divided into one or more frames for transportation over TCP connections.

#### Standardized Wire Formats

Today’s most prominent open standard for messaging is the Advanced Message Queuing Protocol (AMQP) [297]. The AMQP 1.0 transport model is an OASIS standard [193] that recently became the international standard ISO/IEC 19464 [126]. The transport model specifies a binary wire format that multiplexes channels into a single TCP connection or SCTP association, supports flow control for messaging, authentication [161], and encryption by TLS. AMQP is stateful because communicating peers negotiate a session with a handshake when the TCP connection has been established. Peers exchange so-called frames that contain header fields and binary content. For interoperable representation of messages, AMQP offers a self-contained type system. The type system includes a set of primitive datatypes, descriptors for specifying custom types, restricted datatypes, and composite types for structured information. This allows self-describing annotated content when interaction between heterogeneous platforms takes place.

Known as Jabber, and initially motivated by portable instant messaging, the Extensible Messaging and Presence
Fig. 11 The survey distinguishes a web- and service-oriented view on architectures.

Protocol (XMPP) [261, 262] is a standard for messaging services. Short text-based messages, i.e. XML stanzas, are bidirectionally exchanged in open-ended XML streams over a long-lived TCP connection, eventually protected by TLS, between a client and service or service-to-service. To overcome middleboxes, XMPP can utilize HTTP and push technology, i.e. BOSH [225]. Also, XMPP over WebSocket is in an experimental state [281].

The text-based Streaming Text Oriented Messaging Protocol (STOMP) [279] is also an interoperable messaging protocol that has resemblance to HTTP. It operates on top of a bidirectional byte-stream-based transport protocol such as TCP or WebSocket, supports TLS for encryption, and uses UTF-8 as default character encoding.

MQ Telemetry Transport (MQTT) [112] is an open standard for a lightweight messaging protocol on top of TCP and low bandwidths as encountered in Internet of Things ecosystems. It features a binary message format, has a small fixed-size header of only two bytes and therefore little overhead. MQTT also supports SSL/TLS for encrypted transfers.

The Data Distribution Service for Real-Time Systems (DDS) [201] is a machine-to-machine middleware specification. DDS also specifies the Real-Time Publish-Subscribe (RTPS) [202] binary wire protocol for TCP- and UDP-based messaging, including IP multicast for One-to-many Send interaction.

5 Architectures

An architecture integrates protocols, languages, and service interaction patterns for service delivery. Figure 11 indicates the structure of this section. In this article, architectures are distinguished into a web- and service-oriented view. While the web-oriented view explores architectures for typical web scenarios, the service-oriented view focuses on architectures known from enterprise service integration with applicability in cloud computing. This section surveys prominent architectures in both groups, introduces architecture-specific protocols when necessary and refers to software implementations when appropriate.

5.1 Web-Oriented View

The World Wide Web is all about hypermedia: to present multimedia content like text, audio, and video in a non-linear way, and users can access information through hyperlinks. The web has evolved through several phases that are typically referred to by buzzwords [66].

While Web 1.0 refers to the advent of accessible hypermedia, contents were often static, non-interactive and created only by few [55]. The key aspects of Web 2.0 are technological advances for dynamic content, interactive user interfaces, and also a social community aspect; users can interactively collaborate and create new hypermedia content. Some examples are blogs, electronic marketplaces, or social networking. Web 3.0 is believed to add personalization based on semantics of content, for example, adaptivity to user preferences and context. The user experience in discovering knowledge is expected to go beyond merely following hyperlinks and include social, mobile, and location aspects of the user.

Web applications are discussed as a reference architecture for hypermedia delivery, web syndication and mashups as a form of composition.

5.1.1 Web Application

A web application, hosted on a webserver, delivers websites to clients. The fundamental elements of a website are summarized in Figure 12. The resources are uniquely identified by URLs and a client, i.e. a web browser or user agent in general, requests them from a web application over HTTP or HTTPS.

- A website composes text content and media resources using HTML or XHTML markup. A client retrieves the markup and parses it into a DOM tree.
- The markup can define or refer to a CSS resource that specifies the visual representation of the DOM, e.g. colors, fonts, images, or effects.
- The JavaScript runtime environment executes nested or embedded script code, if present, in context of the DOM to interactively modify the tree structure during runtime.
The DOM is then eventually rendered in a window.

For composing multimedia and text, tags in the markup can refer to other resources via URLs to embed them. Depending on the content type of the resource, either predefined in the markup or announced through HTTP, the client either parses and presents the resource natively or hands it to an appropriate plugin during runtime. The HTTP optimizations discussed in Section 4.4.3 aim to minimize the delay experienced by the user when numerous resources need to be loaded to compose a website.

After loading, the user has the choice of non-linear continuations through hyperlinks. A hyperlink refers in fact to a URL, and when followed, a full page transition is triggered, the current DOM is replaced by a new one, and embedded resources are recursively loaded again. As common in Web 1.0, if a resource changes on the service-side, the client needs to actively poll for changes, construct a completely new DOM and reload all uncached resources.

For statefulness between page transitions, web applications use a unique identifier in the URL, in hidden form fields, or rely on the cookie mechanism of HTTP to associate a client request to a service-side session to track application state. This allows, for example, an authenticated session spanning over several page transitions.

In terms of interaction patterns, a traditional web application inherits bilateral Send-Receive from its application layer protocol HTTP or Multi-Responses in case of audio or video streaming over HTTP.

**Browser Content Policies.** Central to the security model of web applications is the *same-origin policy* on the client-side. In general, the DOMs of markup documents retrieved from different origins are isolated from each other for security reasons, including the access and transmission of cookies. The same-origin policy applies to network accessible resources and script APIs [178]. While the policy allows outgoing write access, e.g. form submissions, and static embedding of foreign resources in markup, read access during execution of script code is only allowed to resources from the same origin. Access to a DOM and its APIs through script code is also restricted. Script code is executed in context of the DOM that embeds the script code. If two DOMs that share the same origin execute two scripts simultaneously, both scripts can freely access each other’s DOMs and APIs. The same-origin policy enforces a coarse access control in web browsers to isolate and separate interaction between simultaneously active websites. Cookies are automatically added to HTTP request headers if available for some origin.

The same-origin policy complicates websites that use script code to dynamically compose resources from different origins. A recent enhancement in browser security for a more fine-grained access control to foreign resources is achieved by the technique Asynchronous JavaScript and XML (AJAX) [87]. AJAX relies on the XMLHttpRequest API on the client-side to initiate HTTP or HTTPS interaction from script code during runtime.

Instead of delivering a large HTML markup content that requires a substantial time for parsing and resolving embedded objects, an AJAX-enabled web application typically serves only a small markup skeleton and nested script code, so the client can asynchronously request the slower-loading elements and update the DOM when available. AJAX updates can also be triggered by client-side events such as user interface events. HTTP push technology, as discussed in Section 4.4.3, e.g. Comet or WebSocket, furthermore enables asynchronous near-real-time updates from the service to the client. Therefore, user-experienced page load times and interactivity improve.

AJAX explicitly refers to XML as format for updates, but the mechanism can in fact accept any text-based content type, e.g. JSON, and process it through JavaScript functions during runtime.

![Fig. 12](image-url) The core elements of a website: HTML/XHTML markup structures text and media resources; CSS defines visual representation, eventually by embedding media; and JavaScript provides interactivity.

**Distributed Authoring and Versioning.** One of the earliest attempts to allow users to collaborate over the web is the Web Distributed Authoring and Versioning (WebDAV) [63] extension for HTTP. While the first evolution of the web has offered read-only content for most of the users, WebDAV adds write access such that users can edit resources together. This is accomplished by adding new methods and header fields to the HTTP/1.1 standard, therefore both client and service have to implement this extension.

Two well-known and widely used extensions of WebDAV are CalDAV [58] for shared calendars and task lists and CardDAV [57] for sharing address books.

**Asynchronous JavaScript and XML.** A characteristic development of Web 2.0 is towards dynamic web applications, where, instead of full page transitions, only parts in a DOM are updated during runtime for better user experience. This is achieved by the technique Asynchronous JavaScript and XML (AJAX) [87]. AJAX relies on the XMLHttpRequest API on the client-side to initiate HTTP or HTTPS interaction from script code during runtime.
The same-origin policy also applies to AJAX. Accessing networked resources through the XMLHttpRequest interface is considered a read access and only allowed to resources that share the same origin. A standardized procedure called Cross-Origin Resource Sharing (CORS) [333] allows an XMLHttpRequest access to a foreign resource, but this access has to be explicitly approved by the foreign web application through the use of HTTP headers.

For cross-origin access to JSON resources, when a client does not support CORS, there exists a workaround by script embedding, called JSON with Padding (JSONP) [123]. The same-origin policy does not apply to resources statically embedded in markup. A JSON document is valid JavaScript code, and JSONP exploits the script tag to load the JSON document nested in script code from a specific URL. This script code then calls a user-defined function in the context of the DOM to process the JSON object.

Semantic Web. Information in the web is primarily encoded in semi-structured HTML or XHTML documents, often using natural language, which turns discovering, sharing, and combining information into a hard problem. The Semantic Web is an attempt to standardize a number of formats, so the “web of documents” can become a machine-interpretable “web of linked data” [331]. Linked data is believed to be a major characteristic of the Web 3.0.

The W3C-proposed Semantic Web has three technological pillars: XML and Unicode as fundamental language; the Web Ontology Language (OWL) [326] for specifying relations; and Resource Description Framework (RDF) [309] as a collection of specifications including a metadata data model, vocabularies, and serialization formats.

Several standards have been proposed to increase applicability of W3C’s Semantic Web in today’s document-based web. RDF through attributes (RDFa) [330] is an extension to HTML and XHTML, where semantics of existing markup are annotated by special HTML or XHTML attributes. Microdata [329] for HTML5-based documents and JSON for Linking Data (JSON-LD) [336] are other annotation-based approaches; both standards are compatible with RDF. Microformats [162] is another annotation-based approach, but independent from W3C Semantic Web specifications.

HTML5 and Assisting Technologies. The numerous versions of HTML and XHTML, and ambiguities in their specifications have led to incompatible implementations or varying visual representations in web browsers. Rich web client applications relied on plugins, e.g. Adobe Flash, which had a negative impact on the accessibility for many clients when a plugin was not supported.

Today, HTML5 is an attempt to define an unambiguous syntax for a unified reference markup language and also standardize APIs for a number of assisting technologies that have contributed to the success of Web 2.0 in building rich web applications in the last years [335], such as:

- Semantic annotation of markup through Microdata;
- A set of natively supported audio and video formats;
- App Cache for offline storage of HTML5 websites;
- File System API for persistent storage;
- Web Storage and IndexedDB for temporary and offline storage of client data;
- Concurrent execution of script code by Web Workers;
- Web Messaging, also called cross-document messaging, for controlled data exchange between DOMs of different origin;
- Server-Sent Events (SSE) and WebSocket to increase interactivity of websites that require high responsiveness;
- Geolocation access for localized personalization.

There are also non-W3C technologies that aim to improve capabilities of the client, in particular, WebGL [138] for 3D rendering support, and WebCL [139] for using parallel computing hardware on the client. Also, WebRTC is available through a script API in many modern browsers today, but not yet part of the HTML5 specification. HTML5 is an evolutionary step towards accessible websites and rich web client experience across web browsers and mobile devices.

5.1.2 Web Syndication

While a web application is by default a single and independent service, a client needs to actively interact or rely on HTTP push technology to recognize service-side updates. The goal of syndication is to propagate notifications about updates or push content to clients and also other web applications for service-to-service interaction, e.g. blogs.

The simplest form of change notification is a service-side webhook [152]. There are two parties: a peer that experiences an event and triggers an HTTP request to a special webhook URL, and a routine that handles requests and processes the posted information for the webhook URL. The definition of a webhook is kept abstract, there is only a trigger and a handler in the spirit of callback or event programming. Lindsay [153] further argues that callback principles in the web will eventually lead to the Evented Web, where, in terms of service interaction, routing patterns and messaging between services become feasible. An example PaaS that relies on webhooks for distributed asynchronous processing is Google’s App Engine [94].

Another approach to syndication is by so-called feeds or channels. Clients and web applications can subscribe to a feed offered by a syndication service to receive updates, i.e. a service-side One-to-Many Send interaction pattern. Two standards for feeds are the Rich Site Summary (RSS) [255] and the Atom syndication format [182]. Both are based on
XML. While RSS assumes HTTP as transport, Atom defines it’s own HTTP-based publishing protocol (AtomPub) [98].

A syndication service needs to initiate communication with the client when updates are available. A web client can either poll the syndication service’s feed or use HTTP push technology. For service-to-service interaction, a web application registers a webhook at the syndication service to receive updates when available. A syndication service that pushes updates to a set of clients conceptually takes the role of a broker in a publish-subscribe architecture as shown in Figure 17. Two notable implementations for syndication services are PubSubHubBub [243] and the AtomPub-oriented Apache Abdera [8].

The Bayeux protocol [258] is a push technology framework based on Comet using named channels for a broker-based publish-subscribe architecture, i.e. One-to-Many Send interaction. Clients subscribe to named channels, and the server pushes updates to all registered clients. A potential field of application is web feeds.

5.1.3 Web Mashup

A mashup composes so-called web components, for example multimedia resources or script code, from different origins into a new website [66]. Examples for mashup components are JavaScript libraries, gadgets [90] but also services like Google Maps. Web components and the mashup principle achieves composability and reusability which is also a contributing factor for the success of the Web 2.0. A mashup service, also called integrator, can be distinguished based on the location, where integration takes place [259,274]:

1. **Service-Side Mashup.** When a client requests a mashed-up service, the service first gathers the foreign resources from other origins, processes them, and returns the integrated markup and media to the client.
2. **Client-Side Mashup.** A client-side mashup service returns only a markup skeleton that embeds components or dynamically loads them using AJAX. The client is then responsible to gather web components and integrate them.

Figure 13 shows both cases. The script code acts as a glue between components and the resulting information flow can cause security issues. Web components therefore require proper encapsulation [155]. In terms of patterns, a mashup enables one or more simultaneous bilateral Send-Receive interactions between a client and several web servers that host the integrated web components, eventually routing messages between them.

Content policies lead to two extreme cases of script code interaction: no separation nor isolation by embedding in the same DOM and runtime using `object` or `iframe` tags. Ryck et al. [259] survey the state-of-the-art techniques for more fine-grained controls in mashups and they distinguish four categories of web component integration restrictions:

1. **Separation and Interaction.** Components are separated such that individual component DOMs or DOM parts and script code, e.g. global variables, are isolated from each other. Components can then interact through specified channels. For example, HTML5 offers the `sandbox` attribute for fine-grained `iframe` separation and Web Messaging for interaction between `iframe` instances.
2. **Script Isolation.** To isolate script code in components when executing in the same runtime environment, code is often restricted to a subset of allowed JavaScript functions and static checking enforces an isolation policy.
3. **Cross-Domain Communication.** The same-origin policy denies cross-domain communication for script code by default during runtime. Techniques like CSP, CORS, or an XML `HttpRequest` proxy enable read access according to some policy.
4. **Behavior Control.** This category subsumes techniques to enforce policies on script code execution during runtime, for example by reference monitors, access mediation to script objects, or information flow control between components.

To ease composability of mashup components, public API specifications, also referred to as Open APIs, have become popular in the last years. Such Open APIs can range from simple resource access to sophisticated service architectures as discussed in Section 5.2. In particular, OpenSocial [212] is an initiative to standardize APIs for building social applications that run on the web. Integrating the social dimension is a step towards personalized user experience, which is a characteristic to the Web 3.0.

5.2 Service-Oriented View

Application layer protocols for enterprise services are eventually not forwarded correctly over the Internet because middleboxes might filter them. By using web technology for
communication in service-oriented architectures, for example in a middleware approach, access across the Internet is ensured.

In our definition, a service has an interface that accepts and responds with a certain language. According to characteristics of the interface, services can be further distinguished by:

- **Static typing.** When the accepted language of a service is predefined by an Interface Definition Language (IDL) or a grammar, e.g. a restricted content type, a schema in XML, or a specification in ASN.1, the interface is strict. An implementation can be automatically generated for the accepted language, e.g. a parser or stub code.

- **Dynamic typing.** On the other hand, a service eventually accepts a set of content types, a language family, or messages carry their own specifications, e.g. embedded schemas. Interpretation is then runtime-dependent, and parsers have to be more general or modular. Such a service interface is referred to as dynamically typed.

Typing affects coupling between clients and services. Web applications, syndication, and mashups in the previous section are dynamically typed because the content type of a resource governs the selection of a proper parser and consequently interpretation in the client. Web applications are therefore loosely coupled.

Statically typed service interfaces restrict flexibility and interoperability, for example the allowed programming languages, where stub code is available. The consequence is tighter coupling. Services and middleware can still benefit from loose coupling because evolving software systems becomes easier and cheaper to integrate [230].

This section surveys architectures for offering services with respect to web and cloud ecosystems. Four architectures are discussed: Remote Procedure Calls (RPC), “big” web services, RESTful web services, and message-oriented middleware.

### 5.2.1 Remote Procedure Calls

RPC is a simple yet powerful architectural style to offer a service by exposing network-accessible functions. In terms of patterns, RPC is the bilateral Send-Receive interaction between a client and a service as shown in Figure 14. The client initiates the interaction, and if not stated otherwise, a network function call in RPC is typically synchronous and interfaces are statically typed. Specifying an RPC service requires an agreed-upon transport mechanism, e.g. a TCP connection or HTTP, an agreement how to address and bind to a remote function, and a data serialization format to exchange structured data, e.g. ASN.1. Historically, one of the most widely deployed RPC solutions is Open Network Computing RPC (ONC-RPC) [286], e.g. for network file systems.

**Fig. 14** RPC is conceptually a Send-Receive interaction. A client serializes the arguments for a remote function call as a message, sends it to the RPC service, and waits for a response.

ONC-RPC originates from Sun Microsystems and APIs are available on practically all major platforms. ONC-RPC relies on TCP and UDP as transport mechanism, where call and return values are serialized in the XDR format.

RPC for distributed systems has evolved from function calls to distributed computation over shared objects [300]. Zarras [350] analyzes three prominent RPC-style middleware approaches that are based on object sharing:

- Microsoft Component Services (COM+) [166],
- The OMG-standardized Common Object Request Broker Architecture (CORBA) [203],
- Java Remote Method Invocation (RMI) [217] in the Java Platform, Enterprise Edition (Java EE) [215].

All three approaches define individual data serialization formats and support communication over the Internet protocols, in particular, TCP. While protocols and wire formats for COM+ are specified in the DCE/RPC standard [285], CORBA uses the Internet Inter-ORB Protocol (IIOP) [204] for communication over TCP connections. Java RMI specifies its own native protocols and wire formats on top of TCP, e.g. the Java Remote Method Protocol (JRMP) or Oracle Remote Method Invocation (ORMI), but also supports RMI over IIOP [218] for compatibility with CORBA systems.

The three approaches enable shared objects in an RPC-style architecture. An object broker for allocation, garbage collection, and transactions is implicitly required [300]. To represent a shared object on the client-side, a stub abstracts away the serialization and communication. The service interface is therefore statically typed. A recent middleware framework, similar to CORBA but compatible with the web, is the Internet Communications Engine [351].

**XML- and JSON-RPC.** A trivial architecture for a web-based RPC service is XML-RPC [269]. XML documents serialize call and return information as text-based XML format, and HTTP transports request and response documents to a service identified by a URL. While XML-RPC restricts documents by exactly specifying rules how datatypes are serialized using tags and attributes, an XML-RPC service is
still dynamically typed because there exists no schema for an actual service interface. The XML-RPC Description Language (XRDL) [346] is an attempt to define XML-RPC services in the spirit of an IDL, so client-side stub code can be automatically generated.

JSON-RPC [176] is related to XML-RPC, with the difference that the specification does not restrict itself to any transport mechanism. JSON-RPC only specifies call and return formats based on the JSON format.

Apache Avro, Etch and Thrift. Several proprietary implementations for RPC-style service interaction have been proposed, especially for high-performance provider-side back-end services in clouds. Apache Avro [13] is both a data serialization format and an RPC mechanism in the Apache Hadoop [16] project for big data analysis. Transportation of messages in Avro is protocol-independent in general, but the specification explicitly refers to HTTP. Avro supports dynamic typing: a serialized message is accompanied by its schema in JSON format.

The Apache Etch [11] framework originates from Cisco Systems. It defines a service description language, similar to an IDL, a binary serialization format, and TCP is assumed for transportation. A compiler generates stub code for the client- and service-side of a specified architecture, and messages are therefore statically typed.

Apache Thrift [10], developed by Facebook, is a framework that specifies an IDL for RPC-style services and tools for stub code generation. While transportation and serialization of messages is kept abstract, Thrift already provides several default procedures, e.g. serialization in a compact binary format or JSON and transport over TCP or HTTP. A notable RPC framework integrating Thrift is Twitter Finagle [293]. Finagle furthermore enhances Thrift with multiplexing; Twitter’s Mux session-layer protocol is situated between TCP and Thrift, so only a single connection between client and service is necessary to handle multiple Thrift RPC interactions.

Protocol Buffers. Google Protocol Buffers [89] define a platform-neutral language for specifying a data structure, similar to ASN.1. Tools translate a specified structure into efficient stub code for compact binary content serialization and deserialization. RPC services using Protocol Buffers are therefore statically typed. There are no restrictions on transport mechanisms or message exchange. As long as both parties have stub code generated for the same specified structure, contents can be exchanged, for example over HTTP. Google claims that their internal RPC services rely on Protocol Buffers [89].

Hessian and Burlap. Hessian [47] is a protocol for RPC-style web-based services, where structured information is serialized in a compact binary format. The message format is dynamically typed, designed for efficient processing, and intended for HTTP transportation. No IDL is therefore necessary for a Hessian service. Burlap [46] is conceptually the same protocol as Hessian, but uses an equivalent XML-based data serialization format.

5.2.2 SOAP/WS-* Web Services

Many RPC architectures require static typing in an IDL, and as a consequence, tight coupling through code restrictions. The goal of so-called “big” web services is to relax this coupling by open standards for heterogeneous platforms [230]. A web service deals with XML documents and document encapsulation to evade the complexity of distributed computation on shared objects [300]. The core technology in this attempt is the Simple Object Access Protocol (SOAP) [317] for expressing messages as XML documents.

SOAP is transport-agnostic by design and supports all kinds of transport mechanisms, including message-oriented middleware discussed in Section 5.2.4. HTTP has nevertheless become an industry standard for web and middlebox compatibility [96]. Web services standards (referred to as WS-*) extend SOAP-based interaction with security, reliability, transaction, orchestration, workflow, and business process aspects.

Technologies. In accordance to [4], technologies for web services are categorized into four groups of purpose:

- **Service Description.** The XML-based Web Services Description Language (WSDL) [305] is a format to describe web services similar to an IDL. While coupling between web services is loose due to XML, interfaces and messages in the SOAP/WS-* stack are nonetheless statically typed through WSDL.
A WSDL version 2.0 document has an abstract and a concrete section as shown in Figure 15. The abstract section defines: a type system of one or more XSDs to specify message formats; one or more interfaces and their operations; and an assignment of input and output message types to operations. Every operation has a message exchange pattern: in for Receive, out for Send, and in-out for Receive-Send service interaction.

The concrete section in WSDL describes how abstract interfaces become network-accessible. A binding associates a specific transportation mechanism, e.g. SOAP over HTTP, to an abstract interface. Finally, a service implements an abstract interface by specifying a set of endpoints. An endpoint associates an accessible address (URL), where the service can be consumed, to a concrete binding.

- **Service Discovery.** Services described in WSDL need to be published to enable, for example, automatic discovery or dynamic late binding. The XML-based Universal Description, Discovery and Integration (UDDI) standardizes a registry for classifying, cataloging, and managing web services. Such a registry is typically offered as a SOAP/WS-* web service.

- **Service Interaction.** To access operations in services, clients and services need to communicate SOAP messages. Alonso et al. [4] introduce the notion of a service interaction stack as shown in Figure 16. The stack has four layers: transport of messages by HTTP, and eventually securely transferred using SSL/TLS; messaging through SOAP; a protocol infrastructure to coordinate a number of services using meta-protocols; and middleware properties with respect to security, reliability, transactions, and orchestration of services.

- **Service Composition.** Network-accessible operations allow web services to be composed. In terms of interaction patterns, a basic web service that computes a result implements Send, Receive or Receive-Send. A composite service consumes other services, and according to the application logic, it allows more advanced interaction patterns.

Due to the large number and many versions of WS-* standards, the Web Services Interoperability Organization (WS-I) establishes best practices for interoperability, published as Basic Profiles [339,340]. Figure 16 is limited to Basic Profile protocols and standards for transport and messaging.

**SOAP Attachments.** A SOAP message is structured as follows. The root element is an envelope that holds a header element for metadata and a body element for the message content. The structure of the body has to obey the schema of its according WSDL message type, and a service can therefore validate messages.

Base64 allows to encapsulate arbitrary binary data as XML-compatible text, but incurs a blowup in size and relevant metadata such as the MIME type of the binary content is not preserved in a standardized way. Several techniques have therefore been proposed for dealing with binary content in SOAP.

Two historical standards are SOAP Messages with Attachments (SwA) [304,310] and Microsoft’s DIME. SwA relies on MIME multipart as container format, where XML-based SOAP is the first part, and subsequent parts hold the binary contents of individual MIME types. DIME operates in the same spirit but with a different container format. As there are different types of MIME multipart containers for SwA, the WS-I has published the Attachments Profile [338] for interoperability.

The state-of-the-art SOAP Message Transmission Optimization Mechanism (MTOM) [315] extension resorts to XOP packaging if a WSDL input message type defines an element annotated with a certain MIME content type. So, the client implicitly becomes aware that MTOM is accepted as input format. If a client does not support MTOM, Base64 is used as fallback encoding.

**Messaging.** The core SOAP specification does not standardize how messages are routed, whom to respond to when communication is asynchronous or where to report errors. SOAP is in a sense similar to RPC; the operation is completely defined by its endpoint address, for example, a URL for a HTTP transport binding.
WS-Addressing [311] specifies two core constructs to enable routing patterns in terms of service interaction: endpoint references, and message addressing. An endpoint reference is an address, i.e., URL, and optional parameters for communicating with the endpoint. Addressing information is then placed in the SOAP header of a message, for example, a semantic message identifier (wsa:Action), destination endpoint reference (wsa:To), routing information (wsa:ReplyTo), or error handling (wsa:FaultTo). Based on addressing information, a service can then dynamically forward SOAP messages to other services for various interaction patterns.

WS-Eventing [324] and WS-Notification [185] are two competing standards for publish-subscribe messaging on top of WS-Addressing. In terms of interaction patterns, publish-subscribe is One-to-Many Send from point of view of a publisher. Publish-subscribe either needs a broker service or the publisher manages subscriptions individually, as shown in Figure 17. A broker stores subscribed endpoint references and distributes messages from publishers. So, broker-based publish-subscribe allows a publisher to address an anonymous group of receivers.

Security and Reliability. SSL/TLS in HTTPS bindings for SOAP only ensures encryption between the client and the endpoint of a service, but according to WS-Addressing, a SOAP message can traverse several services or brokers until its destination is met. WS-Security [186] defines standards for end-to-end cryptography methods, including XML signatures and encryption for SOAP messages. WS-Trust [196] and WS-SecureConversation [192] extend WS-Security by establishing a security context for communicating parties to speed up the cryptographic key exchange.

The WS-Policy [318] framework for web services defines a language for specifying and advertising policy assertions, for example, security or Quality-of-Service policies. WS-SecurityPolicy [195] is an extension of WS-Security to express security-specific policies.

Another issue that arises in a messaging context is reliable message delivery. WS-Reliability and its successor WS-ReliableMessaging [191] specify procedures, so delivery is assured through acknowledgements even for routed SOAP messages.

Protocol Infrastructure and Middleware Properties. To exploit SOAP messaging in SOA, standards for coordinating and composing services for business processes are required. A notable standard for such a protocol infrastructure is WS-Coordination [190] for coordinating actions between web services. WS-Coordination enables distributed activities like transactions (WS-AtomicTransaction [188]) or business activities (WS-BusinessActivity [189]).

With respect to standards for composition and orchestration, we direct the reader to the survey of Beraka et al. [32]. Examples for composition and orchestration modeling are the Business Process Model and Notation (BPMN) [205], Business Process Execution Language (BPEL) [187] and WS-Choreography [316].

Software Implementations. In the Java world, there are several APIs supporting SOAP/WS-* web services integration like JAXM [129] for SOAP integration, the obsoleted JAX-RPC [128], and its successor JAX-WS [130] for web services. The most notable implementations using those APIs are Apache Axis2 [9], Apache CXF [14], GlassFish [132], IBM WebSphere [110], JBoss [222], Oracle Weblogic [219], and XML Interface for Network Services (XINS) [97].

Furthermore, the Windows Communication Framework (WCF) [170] is an API and runtime environment in Microsoft’s .NET framework that relies on SOAP/WS-* for services integration.

5.2.3 RESTful Services

Fielding [75] has coined the term “Representational State Transfer” (REST) as an architectural style for distributed hypermedia systems in the last decade. REST for resource-oriented architectures has become a key technique in the Web 2.0 and for cloud services to achieve simplicity, scalability, and sharability. In its original form, REST is rather abstract and not limited to specific protocols. Nevertheless, the principles have been derived from successful web architectures and early versions of HTTP, that is why REST primarily refers to HTTP as transport mechanism today.

REST emphasizes a unified interface between components and abstraction of information as resource. Four interface constraints for an architectural style to be considered RESTful are established by Fielding [75,77]:

1. Identification of Resources. A REST service offers a set of resources that need to be identified, so clients can interact with them. URIs are a well-known technique for
resource identification in the web and also for REST. HTTP is the protocol of choice for interaction.

2. Manipulation of Resources Through Representations.

A representation of a resource is a sequence of bytes plus metadata that describe those bytes using a content type. A representation captures the current state of a resource, and REST operations applied to a representation modify the resource’s state. REST reuses HTTP methods as unified operations on resources. So, existing infrastructure such as caches or proxies stay compatible with REST, which in turn allows high scalability as needed in PaaS or SaaS. The following HTTP methods are defined as operations:

- Method PUT creates a new resource from a transferred representation.
- The idempotent method GET reads a representation of a resource’s current state.
- Method POST updates the state of a resource to the transferred representation.
- Method DELETE deletes a resource.

3. Self-Descriptive Messages.

A resource can be represented in various formats, e.g. HTML, XML, or JSON, so a client can choose or negotiate a viable representation. On the other hand, a resource’s metadata allows decision making for caching, content negotiation, checksums, but also authentication and access control [232].

4. Hypermedia as the Engine of Application State.

Interaction with a resource is stateless and, in terms of interaction patterns, Send-Receive for a client. A RESTful service only manages resource state, and the client is responsible for tracking application state as shown in Figure 18. A pure RESTful service in Fielding’s definition therefore has no notion of session as common in web applications; every Send-Receive interaction has to be self-contained.

A client accesses a RESTful service through a single entry point, i.e. a bookmark, and the service returns hypermedia as simultaneous presentation of information and control [76]. The hypertext outlines potential choices a client can take in a specific state; choices are basically hyperlinks that point to continuation URIs. Interpretation of a resource’s representation then depends on its content type. Application state is therefore completely on the client-side, and REST is an extreme case of dynamically typed interface. The hypertext-driven presentation enables loose coupling, maximum freedom in resource namespaces, and allows dynamic discovery of resources on the client-side.

There has been an extensive debate in literature on REST versus SOAP/WS-* over the years [230, 231, 241], including a quantitative comparison by Pautasso et al. [232] with respect to degrees of freedom. While SOAP/WS-* standards precisely define how to implement certain properties of a service, REST is a number of principles that outline characteristics that a service needs to satisfy. This freedom-of-choice in REST has led to many services that claim to be RESTful but are in fact RPC [76].

Technologies. In comparison to SOAP/WS-* web services, technologies for REST can also be grouped into four categories of purpose:

- Service Description. APIs for RESTful services are often just documented in natural language on a website, e.g. Open APIs for mashups. There are two attempts in machine-interpretable RESTful service descriptions: WSDL 2.0 [305] and the Web Application Description Language (WADL) [321]. The latest version of WSDL supports non-SOAP messages and has more fine-grained control over HTTP bindings to turn them RESTful. WADL is a description language for HTTP-based web applications to enable modeling or automatic stub code generation for RESTful service APIs. WADL is XML-based and allows to describe a set of resources, relationships between resources, available methods, and representations for a resource. Furthermore, WADL supports XSD and Relax NG as schema languages to restrict XML-based resource representations.
- Service Discovery. Resource discovery in REST is dynamic due to the “Hypermedia as the Engine of Application State” principle. In Figure 18, for example, the cart resource is the entry point and provides hyperlinks to products, so the client discovers its choices and proceeds to the next internal application state. Discovering a service is then uncovering an entry point, i.e. bookmark or hyperlink. Entry points can be managed in registries accessible as a service, e.g. Google APIs Discovery Service [91], announced through publishing protocols, e.g. AtomPub, or published through DNS-based service records [50].
- Service Interaction. Formats of SOAP/WS-* messages are inherently XML-based and specified in a WSDL.
REST does not restrict representations of resources and allows free usage of content types. Some popular text-based formats for data serialization in RESTful services are JSON, plain-old XML (POX), and YAML. As long as metadata refers to a MIME type and a client can understand the representation, it is a valid format for REST. REST relies on client-service interaction, but does not standardize any kind of multilateral messaging like WS-Addressing in SOAP-based web services. A typical application of REST in context of messaging is as a web-accessible API to message-based middleware (discussed in Section 5.2.4).

- Service Composition. Today, composition of RESTful services primarily takes place in web mashups, where resources serve as web components. There are also attempts towards composition in JOpera [227] and workflow orchestration by BPEL for REST [228], BPMN for REST [229], or the JavaScript-based 3 language [36].

Hi-REST and Lo-REST. Pautasso et al. [323] distinguish HTTP-based implementations into Hi-REST and Lo-REST: Hi-REST uses the four HTTP methods for operations, POX for data serialization, and resources have “nice” URIs. But many web browsers are limited to HTTP methods GET and POST which restricts available REST operations when integrated in web applications or mashups. Lo-REST deals with these restrictions and, as a workaround, exploits an HTTP header or form field to store the actual operation that needs to be applied. This can of course affect caching or proxy infrastructure.

Security and Reliability. For secure exchange of representations, REST relies on SSL/TLS in HTTPS. Only the connection between client and service is secured this way, there is no standardized way for end-to-end security in a messaging context, like WS-Security. Furthermore, REST over HTTP has no standardized form of reliable delivery or transaction handling compared to SOAP/WS-* web services.

Software Implementations. REST is more an architectural style than a set of technologies, and RESTfulness is achievable in all kinds of web frameworks or programming environments, for example, Google Gadgets for mashups [90].

Specifically in the Java world, the JAX-RS API [131] has been introduced for RESTful services, and practically all mentioned SOAP/WS-* software implementations support it too. Other notable implementations and frameworks are the REST extensions in Microsoft’s WCF [170], the PaaS Restlet [252], and the Web Resource Modeling Language (WRML) [159] framework for RESTful API design.

5.2.4 Message-Oriented Middleware

To cope with an increasing demand on scalability, flexibility, and reliability in enterprise and cloud environments, a message-oriented middleware (MOM) is an infrastructure for loosely coupled interprocess communication, for example, for an enterprise service bus [56]. Especially in cloud ecosystems, the loose coupling allows to rapidly scale message producers and consumers. A message with respect to MOM is an autonomous, self-contained entity that models an event and separates into a header and a body or payload. The middleware provides the technical means of exchange, so a peer can send and receive messages to and from other connected peers.

A central concept in MOM is the notion of a message queue (or channel) for storing, transforming, and forwarding messages. Message queues enable asynchronous interaction, and a simple form is a First-In-First-Out (FIFO) queue. There are two different approaches to MOM using message queues as shown in Figure 19:

- Peer-to-Peer Messaging. A unified middleware component in every peer coordinates discovery and interaction between peers.

- Broker-based Messaging. The middleware acts as a broker to provide a messaging layer between the heterogeneous peers.

Peers can participate as client, service, or both [56]. A broker reduces the communication complexity between a number of peers, as demonstrated in Figure 19(b), but can incur delays in real-time applications because an additional store-and-forward procedure is necessary at the broker for message delivery.

In terms of service interaction patterns, a trivial message queue allows bilateral Send and Receive for asynchronous messaging and multilateral One-to-Many Send, e.g. publish-subscribe. Using message queues in a broker architecture allows to implement sophisticated routing patterns. In general, a MOM is characterized by [56]:

- Messaging Specification. A MOM needs to specify the format of messages and transport mechanisms. Intercon-
connecting proprietary MOM systems is achieved through adapters or bridges.

- **Message Filtering.** A core functionality of a MOM is filtering for message delivery. Curry [56] distinguishes:
  - A channel-based system offers predefined groups of events as channels, where clients can subscribe to.
  - Messages in a subject-based system carry metadata in the message header, i.e. a subject. A client subscribes messages, where the subject matches some given string or pattern.
  - In a content-based system, a client subscribes messages, where the message body satisfies a set of properties expressed in a query language.
  - Composite events functionality extends a content-based filtering with property matching across sets or sequences of messages.

- **Message Transformation.** Messages can originate from various heterogeneous sources and consequently carry all kinds of content types as payload. A MOM eventually offers interfaces or operations to modify messages, e.g. XML transformations.

- **Integrity, Reliability and Availability.** A MOM can offer various features to increase the overall Quality-of-Service. This includes:
  - Transactions and Atomic Multicast Notification;
  - Reliable message delivery: at-least-once, exactly-once, or at-most-once;
  - Guaranteed message delivery by acknowledgments;
  - Prioritization of messages;
  - Load balancing over several brokers or queues; and
  - Message broker clustering for fault-tolerance.

A MOM is typically accessed through an API to abstract the technical details of message exchange. Due to the transport-agnostic design of SOAP/WS-* services, a MOM can also serve as a transport mechanism for SOAP messages.

**Java Message Service.** The general purpose API named Java Message Service (JMS) [216] is maintained in a Java community process for MOM support. JMS defines a number of operations for creating, sending, receiving, and reading messages. It is transport-agnostic to abstract messaging from MOM implementations and therefore relays vendor lock-in. The goal of JMS is a universal interface for interacting with multiple heterogeneous messaging systems [56]. A message body is dynamically typed according to the content type information stored in the header.

Some examples for JMS-enabled software implementations are the JMS reference implementation OpenMQ [88], IBM Websphere MQ [111], or TIBCO Enterprise Message Service [287].

**Proprietary Messaging Solutions.** MSMQ [167] is a MOM for standalone integration or as a transport mechanism in Microsoft’s WCF, next to web services and COM+. It offers guaranteed message delivery, message routing, transactions, prioritization, and a simple type system for message body types. When used as a transport in WCF, a message body is either XML, binary, or ActiveX format. Besides its custom protocols, messages can also be transmitted over COM+. In terms of security, MSMQ allows authentication and encryption of messages. There is no broker in MSMQ; similar to Figure 19(a), a queue is hosted locally on a peer, and processes can store and retrieve messages. In terms of service interaction patterns, MSMQ is bilateral Send and Receive. MSMQ can exploit network multicast to replicate a message for addressing multiple queues. An alternative Microsoft solution with brokerage support is SQL Server Service Broker [169].

Other proprietary MOM software products are the brokerless TIBCO Rendezvous [288], which uses direct connections between peers similar to MSMQ, Oracle Tuxedo Message Queue [214] as part of the Oracle Tuxedo application server for cloud middleware, and Terracotta Universal Messaging [272].

**Advanced Message Queuing Protocol.** Historically, MOM solutions have used proprietary protocols, and the transport-agnostic JMS is an attempt to agree on a compatible interface. Interoperability between varying MOM solutions is still difficult; costly JMS adapters or bridges are necessary to connect different transport mechanisms. AMQP [193] is an attempt for unified messaging through an agreed-on wire format and has a similar as HTTP in web applications. While the OASIS AMQP 1.0 standard is restricted to the transport model to achieve interoperability over the Internet, messaging architectures are specified in the preceding specification of the AMQP working group [197].

The AMQP specification distinguishes a transport model (discussed in Section 4.4.5) and a queuing model [209]. The semantic queuing model defines terms like message, queue, exchange, and binding with respect to AMQP. Messages always end up in queues which are analogous to postal mailboxes. A queue stores messages and offers functionality for searching, reordering, or transaction participation. If a client wants to send a message, it chooses a broker-like exchange which is responsible for delivering messages to queues. An exchange can be offered as a service, and there exists an individual URI scheme (amqp:// or amqps://) [235] to locate an exchange. A binding is a set of queue-specific arguments for an exchange. As shown in Figure 20, there are different exchange types with respect to message filtering capabilities [209]:

- In a “direct exchange”, a message has a routing key and is sent to the queue, whose binding is equivalent to the
Technologies for Web and Cloud Service Interaction: A Survey

![Diagram of exchange types]

**Fig. 20** AMQP defines four types of exchanges. A producer creates a message and sends it to an exchange. Depending on the exchange type and bindings, the message is delivered to queues, where consumers can fetch it from.

Routing key. In case of multiple queues with identical bindings, multiple message copies are delivered, i.e. a channel-based system.
- A “topic exchange” forwards copies of a message to all client queues, where the message routing key matches a queue’s binding pattern, i.e. a subject-based system for publish-subscribe delivery.
- In a “fan-out exchange”, messages are forwarded to a set of queues without a specified binding, i.e. channel-based system.
- A “headers exchange” matches the headers of a message against predicate arguments of client queues beyond the routing key, i.e. a content-based system.

Messages are finally fetched from queues by consumer processes. AMQP provides guaranteed delivery, authentication, wire-level encryption, and transaction-based messaging for reliability. In terms of patterns, an exchange applies pattern Send in case of direct delivery or One-to-Many Send in other cases. Due to the self-contained type system and self-describing message content, specified in the transport model, messages are dynamically typed in AMQP.

Examples for JMS-compatible broker implementations are OpenAMQ [114], JORAM [133], WSO2 Message Broker [345], SwiftMQ [282], Apache Qpid [12], and Red Hat Enterprise MRG [248].

**Extensible Messaging and Presence Protocol.** While the XMPP [260,261,262] has been intended as an open standard for instant messaging, presence information, and contact list maintenance in chat applications, it also has middleware applications. In its base specification, XMPP exchanges messages in form of XML stanzas in client-to-service and also service-to-service communication in case of federated services. An XMPP service therefore takes the role of a broker.

XMPP is particular attractive for MOM scenarios, where web agents are involved because it supports HTTP as transport mechanism and most web browsers and JavaScript runtime environments are capable of XML processing. Furthermore, XMPP is also considered as a suitable messaging protocol for Internet of Things applications [347]. The protocol is extensible and extensions are specified in a community process. MOM-specific extensions are:
- Transfer of arbitrary binary content using Base64 and an assigned MIME type [263];
- RPC over XMPP [2];
- Service discovery [101];
- Publish-subscribe [172] for broker scenarios, extended addressing [102] for message routing and event notification extensions [337,264];
- Reliable message transport [173]; and
- SSL/TLS protected transmission and S/MIME [247] for end-to-end message encryption.

By default, messages in XMPP are XML stanzas and bodies are restricted to text only; there exists a notion of message type but it is limited to instant messaging applications. Therefore, out-of-band signaling or a custom protocol, e.g. [263], is required to discover message content types in a middleware scenario.

XMPP can also serve as a messaging infrastructure for SOAP/WS-* web services [80]. Beside instant messaging, XMPP has been successfully deployed in the VIRTUS middleware for Internet of Things applications [54] using the real-time collaboration server software OpenFire [113]. Another software that offers XMPP messaging over WebSocket is the Kaazing WebSocket Gateway [136].

**Streaming Text Oriented Messaging Protocol.** The simple text-based wire protocol STOMP [279] is for asynchronous message exchange between a client and a service or broker with simplicity and interoperability in mind. In the open standard of STOMP, a client and a service establish a session and asynchronously exchange frames of type Message, Receipt, or Error; a frame is partitioned into a command, header fields for metadata, and content of a certain MIME type. Messages are therefore dynamically typed. The protocol supports transactions and acknowledgments for reliable message delivery.

STOMP supports either bilateral messaging, i.e. Send and Receive, or broker-based publish-subscribe as in Figure 17(a) for One-to-Many Send interaction. Two notable service implementations are CoiMQ [53] and, for the latest protocol version 1.2, Stampy [275].
**Message Queue Telemetry Transport.** MQTT [112] originates from IBM and is now an open OASIS standard [198] for lightweight machine-to-machine messaging and Internet of Things applications, where bandwidth is limited. MQTT is intended for broker-based publish-subscribe architectures, as in Figure 17(a), i.e. One-to-Many Send interaction. An MQTT message can transport a binary payload up to 256 megabytes, but there is no notion of content type. The participating parties therefore have to agree on allowed formats out-of-band or use a custom protocol. For reliability, the protocol offers acknowledgments and retransmissions, but there is no transaction functionality.

Two notable MQTT broker software implementations are HiveMQ [103] and Mosquitto [177]. Both support web clients using WebSocket. Another application that relies on MQTT messaging is Facebook Messenger [352].

**Data Distribution Service for Real-Time Systems.** The DDS [201] is another open standard for a machine-to-machine MOM for publish-subscribe message distribution, real-time message delivery, scalability, and high throughput. Fields of application include the finance and defense sector, industry, aerospace, Internet of Things, and mobile devices [291].

Contrary to MQTT, DDS facilitates a data-centric, peer-to-peer interaction in the spirit of Figure 19(a). A domain is the conceptual partitioning unit for entities like publisher, subscriber, and topic. A topic in a domain has a unique name and a strong datatype for publishing; these types are specified in an IDL, and messages are therefore statically typed. Subscribers in the domain request data via the topic, and publishers in the domain are responsible for actual message distribution [200].

DDS allows to express rich Quality-of-Service policies for data transmission. Interoperability between software implementations is achieved through the RTPS [202] wire protocol. To locate endpoints of peers, DDS provides dynamic discovery of publishers, subscribers, topics, and datatypes with respect to topics [291]. Reliable message delivery is achieved by negative acknowledgement when data is missing [200]. Security extensions for DDS are still in a beta state at time of writing [206].

Notable software implementations are OpenDDS [199], RTI Connext DDS [256], PrismTech OpenSlice DDS [242], and Twin Oaks CoreDX DDS [292].

**Apache Kafka.** Developed by LinkedIn, Apache Kafka [17] is a message broker specification and implementation for high-throughput publish-subscribe messaging, i.e. One-to-Many Send interaction. Kafka has an individual binary wire format protocol on top of TCP, and for fault-tolerance, it supports clustering of brokers, persistent storage, and replication of messages.

On a conceptual level, Kafka distinguishes between topics for categories of messages, producers that publish messages, and consumers that subscribe to topics. For every event topic, a Kafka cluster maintains a partitioned log, where every partition holds an ordered sequence of published messages. Messages are stored for a configurable timespan, and partitions are replicated and distributed over servers in the Kafka cluster for fault-tolerance and performance. For every subscribed topic, a consumer maintains an offset in the sequence of published messages to keep track of already processed ones. Through this offset, a consumer can also access older messages if they are still stored on the cluster. The distributed log in Kafka guarantees the ordering of published and consumed messages in a certain topic.

A message body is a byte sequence of a certain length and has no notion of type. Content type information therefore needs to be agreed out-of-band or by using a custom protocol. An interface for web clients to subscribe to Kafka over WebSockets is already in an experimental state [34].

**ZeroMQ.** The intelligent socket library ZeroMQ [115] aims for more flexible connectivity between peers. ZeroMQ offers several network transports, including TCP, UDP, and IP multicast, and a number of sockets types for architectural patterns. Messages are delivered to a thread- or process-local queue and made available through a socket. The specification covers the following socket types:

- REQ and REP for bilateral Send-Receive;
- DEALER and ROUTER for routing patterns;
- PUB and SUB for publish-subscribe One-to-Many Send;
- PUSH and PULL for workload distribution through One-to-Many Send and One-from-Many Receive;
- PAIR for asynchronous Send or Receive between two sockets.

ZeroMQ has no notion of broker because it acts as socket abstraction. Nevertheless, an actual broker for a MOM could be implemented using ZeroMQ. Messages are sequences of bytes and do not have a specified content type. The content type of byte-oriented messages needs to be agreed on out-of-band or requires a custom protocol.

An attempt to provide ZeroMQ access in web environments is NullMQ [154]. The JavaScript library uses WebSockets and a modified version of STOMP to bridge ZeroMQ messages into web browsers.

ZeroRPC [61] integrates RPC on top of ZeroMQ. Information is serialized as JSON-oriented MessagePack format and exchanged over ZeroMQ. A service interface is dynamically typed, and an example application is the Docker PaaS cloud infrastructure.

**RESTful Messaging Service.** The idea behind specifying a RESTful Messaging Service (RestMS) [341] is to turn
messaging truly web compatible by using HTTP as message transport and REST principles to describe locations, i.e., URLs, where messages can be posted to and received from. A web server for brokering the REST interface and client messages is therefore required.

RestMS is an API specification, where XML-based messages are sent and received through HTTP methods. Resource locations are distinguished into feeds for incoming and pipes for outgoing messages with respect to the REST service. Feeds are joined with pipes for message distribution. Message types in RestMS refer to XML, JSON, and a set of MIME content types for dynamically typing data. The specification also defines profiles for bridging to other infrastructures, e.g., AMQP.

**Open Middleware Agnostic Messaging API.** Due to the diversity in middleware standards and wire formats, the Open Middleware Agnostic Messaging API (OpenMAMA) [211] initiative is an attempt to provide a single API for developing applications spanning across multiple MOMs. For correct translation messages and operations, a MOM has to provide a so-called OpenMAMA bridge implementation.

OpenMAMA is available as open-source library. It offers built-in bridge for AMQP-enabled Apache Qpid and supports several bridges for proprietary messaging infrastructures in the finance sector.

**Polyglot Message Brokers.** A natural approach for interconnecting a number of MOM standards is polyglot message brokerage. Three notable JMS-compliant software implementations in this area are Apache ActiveMQ [6], RabbitMQ [236], and JBoss HornetQ [108].

Beside features for scaling and clustering in Apache ActiveMQ, the Apollo messaging core [15] has an individual wire format called OpenWire [7], but also supports other standards like AMQP, MQTT, and STOMP over WebSockets. ActiveMQ offers a proprietary HTTP-based RESTful API to support a wide range of web clients.

RabbitMQ supports AMQP, STOMP, MQTT, and also HTTP as transport. Messages over HTTP can be transported in three ways: a native web application management API; STOMP over WebSockets; and JSON-RPC for web browser integration. HornetQ [108] is a MOM that originates from the JBoss application server. It supports AMQP, offers an HTTP-based RESTful web interface and provides STOMP over WebSockets.

**Message Queuing as a Service.** Message brokerage itself has become an attractive service in the cloud. A broker is a critical component in a MOM architecture and therefore needs proper measures for fault-tolerance, regular maintenance, and scalability; a message queue cloud service can eventually reduce cost. Amazon Web Services offers Simple Queue Services (SQS) [5] for transporting untyped text-based messages up to 256 kilobytes length. SQS operates on a SOAP/WS-* web service stack accessible through HTTP and HTTPS bindings for clients.

Google’s App Engine offers Pull Queues [93] and Push Queues [94] for messaging and task distribution. Both queue types are accessible through a RESTful API and use JSON format for messages. While Pull Queues need to be polled, Push Queues rely on webhooks for HTTP-based message delivery. Google has also announced Cloud Pub/Sub [92], a broker-based publish-subscribe messaging service for the App Engine, cloud apps, and web clients. Using a RESTful API, Cloud Pub/Sub distributes JSON-based messages according to topics. Subscribers can either poll for new messages or register a webhook for notification. The service supports guaranteed message delivery by maintaining a queue for every subscriber, and messages are removed from the queue, when the client acknowledges the message through REST.

Microsoft also offers two cloud-based messaging solutions: Azure Queues, and Service Bus Queues [163]. Azure Queues provide direct messaging between services and they are accessible through a simple RESTful interface. Messages are sequences of bytes and therefore not typed similar to SQS. Service Bus Queues offer more advanced architectures such as publish-subscribe and routing patterns. Windows applications and peers can access a service bus through WCF and HTTP as well. Furthermore, a BrokeredMessage in a Service Bus Queue explicitly denotes a user-specified content type for a message body. Service Bus Queues also offer an AMQP interface [164].

Two other cloud services that offer AMQP-based brokerage as a service are StormMQ [280] and IronMQ [124]. CloudAMQP specifically offers the polyglot broker RabbitMQ as a Service [51]. CloudMQTT [52] is another pay-per-use broker for MQTT messaging, e.g., for complex event processing in Internet of Things scenarios. Rackspace Cloud Queues [244] supports publish-subscribe architectures by a HTTP-based RESTful API in the spirit of RestMS.

### 6 Discussion

Technologies for web- and service-oriented views are converging, especially driven by PaaS and SaaS cloud delivery models. This section discusses several observations drawn from the survey.

**Multiplexing.** There is a trend towards multiplexed transport protocols in the web to minimize overhead when multiple resources are requested in parallel by a web client, especially SPDY and the upcoming HTTP/2. SPDY is already
available in Google Chrome and effectively used as transport mechanism, when Google cloud services are consumed. Multiplexing can also contribute to service-oriented architectures in cloud backend infrastructure, e.g. RPC or MOM, when interactions between two peers occur simultaneously and in large numbers, e.g. the Mux protocol in Twitter’s Finagle [293] RPC framework.

Multihoming. Mobile devices are a growing market, and an increasing number is connected to two or more physical networks simultaneously, e.g. Wi-Fi and 3G/4G networks. Multihoming protocols like MPTCP already exploit this connectivity in Apple devices to increase client-side access quality for the Apple Siri service [19]. Now that implementations and libraries are available on a popular platform, multihoming will eventually become mainstream on other platforms too. Experimental Linux kernels with MPTCP support for Android devices do already exist [294].

Pervasive Encryption. So-called middleboxes for various networking applications, e.g. firewalls or traffic shapers, observe network traffic at neuralgic points all over the Internet and they often rely on DPI as a monitoring technique. Modern transport protocols such as MPTCP and QUIC, but also SPDY explicitly refer to pervasive encryption, so middleboxes that are not aware of these protocols cannot tamper with network interaction. It is safe to assume that encrypted network traffic will flourish in the future, and DPI systems are eventually affected.

Furthermore, an observation in this study is an increasing distrust in X.509 public key infrastructure. Any certificate authority, trusted by a web client, can issue a trusted cryptographic certificate for any service. But certificate authorities are typically shipped with software, so the trust relationship is hardly reviewed. Malign authorities can issue forged X.509 certificates used for man-in-the-middle attacks in network traffic monitoring [109]. An observed counterreaction is a trend towards certificate pinning [223], where a client gets access to additional information for verifying an X.509 service certificate, e.g. a fingerprint of the expected public key or a certificate-specific restrictions on authorities. These techniques are already put to use today, e.g. in mobile phone apps as a protection against reverse engineering [221] or as a security mechanism in the Google Chrome browser [283].

Correct Types. Dynamically typed interfaces offer greater flexibility, but rely on correctness of the content type descriptor of a message, so a correct module for parsing is chosen during runtime. MIME types can be ambiguous because of implicit subtype relations, e.g. every content that satisfies application/xml also satisfies MIME type text/plain but semantics are different. Nevertheless, web technology relies on MIME types as identifiers, and incorrect type information can affect function and security.

Two problems in web browsers with respect to type correctness have been observed. If the Content-Type is missing in an HTTP response, a web browser tolerates the error and tries to guess the MIME type using heuristics, i.e. content sniffing [28]. Furthermore, HTML and XHTML allow to define an expected MIME type as attribute in embed or object markup tags. A type conflict occurs if the specified type and the HTTP response content type are not the same, and the browser or module is responsible for resolving the conflict. Type heuristics and conflicting types have already been exploited to attack web browsers and undermine content policies [28,156]. Correctness of types is therefore essential for reliable and secure composition of web services.

Polyglot Messaging. The diversity of messaging standards for MOM complicate access from web clients. Due to varying application profiles in MOM, e.g. Internet of Things versus enterprise service bus, the diversity of technologies for MOM will likely stay. Nevertheless, there is a trend towards messaging in web and cloud clients for near-real-time communication, so an increasing number of software implementations add web-compatible interfaces like SOAP, JSON-RPC, RestMS, or individual RESTful interfaces. The survey indicates that implementations tend to provide interfaces and adapters for web clients using WebSockets. This transport mechanism allows to connect web clients with messaging standards like AMQP, STOMP, MQTT, and XMPP for direct integration in enterprise ecosystems.

Client-to-Client Messaging. WebRTC has the potential to fundamentally change the web experience. Today, web interaction inherently takes place between a web client and a web application or service. A growing amount of user-provided Web 2.0 content needs to be stored and forwarded, and a lot of effort has been spent to deliver media in time, e.g. Content Delivery Networks (CDNs). Pay-per-use and flexible scalability in cloud computing industrialize this process but governance of data and cloud vendor lock-in is still an unresolved problem for users.

JavaScript and HTML5 technologies turn web browsers into the virtual machines of the web, and WebRTC enables asynchronous interaction directly between them. A potential application is a new era of peer-to-peer networks and code mobility, where clients can seamlessly become service providers, and cloud services merely participate for discovery and signaling between clients. Existing MOM standards could be adapted to provide a messaging infrastructure over WebRTC as transport mechanism. An example for today’s WebRTC usage is PeerCDN [233], a distributed peer-to-peer cache as an alternative to costly CDN infrastructure.
Monitoring Implications. From an interaction aspect, DPI network traffic monitoring [31] is a common technique to enforce some form of security policy, e.g. next-generation firewalls, intrusion detection and prevention systems, data loss prevention, breach detection, censorship, traffic shaping, or middleboxes in general. The survey indicates the following implications about DPI systems:

- To analyze encrypted network traffic, a DPI system has to break the encryption, which is a hard problem or requires a misuse of X.509 public key infrastructure to forge certificates. Otherwise, only metadata about network traffic is available for policy decision making.
- Assuming that a DPI system is capable of breaking the encryption using a man-in-the-middle-attack by misusing an authority to forge certificates, certificate pinning can still prevent actual communication, even when the certificate authority used for forging the certificates is in the client’s certificate chain.
- Furthermore, a DPI system is bound to some physical domain. Multihoming communication utilizes all available physical paths, where eventually one path is invisible to the DPI system, e.g. a path over 3G/4G networks. So, the monitor observes only a fraction of the communication which can impair decision making, even when based on metadata or network traffic characteristics.

A conclusion is that future security systems need to be situated directly on the client- or service-side at communication endpoints or as a component in the client or service. Otherwise, visibility of communication is not guaranteed with upcoming communication protocols.

7 Conclusion

The study presents the state-of-the-art in technologies for interaction between clients and services in web and cloud environments. As clients and services are distributed, communication is necessary for interaction. The survey distinguishes languages, protocols, and architectures: languages encapsulate information in a transportable format; protocols specify languages and rules of engagement to characterize communication; and architectures integrate languages, protocols, and service interaction patterns into blueprints for service delivery.

Architectures are approached in two different views. The web-oriented view highlights the evolution of the web and technological advancements, e.g. AJAX, HTML5, syndication and mashups. The service-oriented view highlights architectures and technologies for services. The survey indicates that technologies in both views are more and more converging.

Languages and protocols that originate from web applications, e.g. HTTP, are becoming an integral part in service and cloud technology because these protocols are supported on a wide range of platforms and correctly forwarded over the Internet. On the other hand, architectures for large-scale distributed systems, e.g. RPC or messaging, more and more influence how web applications are implemented and composed (mashups) to fully utilize the capabilities of an underlying cloud infrastructure. Especially new technologies like WebSocket and WebRTC have a potential to mainstream sophisticated MOM architectures into the web. Furthermore, upcoming protocols and architectures have an impact on network monitoring applications, especially on DPI-based systems. For the future, alternatives like client- and service-centric monitoring approaches need to be considered.

Acknowledgements This research has been supported by the Christian Doppler Society. The author thanks Roxana Holom, Tania Nemes, and Philipp Winter for the valuable feedback to improve this article.

References

1. van der Aalst, W.M.P., Mooij, A.J., Stahl, C., Wolf, K.: Service interaction: Patterns, formalization, and analysis. In: Formal Methods for Web Services, Lecture Notes in Computer Science, vol. 5569, pp. 42–88. Springer Berlin Heidelberg (2009)
2. Adams, D.: XEP-0030: Service Discovery (2011). URL http://xmpp.org/extensions/xep-0030.html. Accessed 2014-07-23
3. Alinone, A.: 10 years of push technology, comet, and websockets (2011). URL http://cometdaily.com/2011/07/06/push-technology-comet-and-websockets-10-years-of-history-from-lightstreamers-perspective/. Accessed 2014-02-17
4. Alonso, G., Casati, F., Kuno, H.A., Machiraj, V.: Web Services - Concepts, Architectures and Applications. Springer (2004)
5. Amazon Web Services: Amazon Simple Queue Service (Amazon SQS) (2013). URL http://aws.amazon.com/sqs/. Accessed 2014-02-21
6. Apache Software Foundation: Apache ActiveMQ (2011). URL http://activemq.apache.org/. Accessed 2014-02-21
7. Apache Software Foundation: OpenWire Version 2 Specification (2011). URL http://activemq.apache.org/openwire-version-2-specification.html. Accessed 2014-02-20
8. Apache Software Foundation: Apache Abdera: An Open Source Atom Implementation (2012). URL https://abdera.apache.org/. Accessed 2014-05-08
9. Apache Software Foundation: Apache Axis2/Java (2012). URL http://axis.apache.org/axis2/java/core/. Accessed 2014-03-28
10. Apache Software Foundation: Apache Thrift (2012). URL http://thrift.apache.org/. Accessed 2014-02-20
11. Apache Software Foundation: Apache Etch (2013). URL http://etch.apache.org/. Accessed 2014-02-21
12. Apache Software Foundation: Apache Qpid (2013). URL https://qpid.apache.org/. Accessed 2014-07-21
13. Apache Software Foundation: Apache Avro 1.7.6 Specification (2014). URL http://avro.apache.org/docs/1.7.6/spec.html. Accessed 2014-02-21
14. Apache Software Foundation: Apache CXF: An Open-Source Services Framework (2014). URL http://cxf.apache.org/. Accessed 2014-03-28
http://firstmonday.org/ojs/index.php/fm/article/view/2125

56. Curry, E.: Message-oriented middleware. In: Q.H. Mahmoud (ed.) Middleware for Communications. John Wiley & Sons, Ltd, Chichester, UK (2005)

57. Daboo, C.: CardDAV: vCard Extensions to Web Distributed Authoring and Versioning (WebDAV). RFC 6352 (Proposed Standard) (2011). URL http://www.ietf.org/rfc/rfc6352.txt. Updated by RFC 6764

58. Daboo, C., Desnoues, B., Dusseault, L.: Calendaring Extensions to WebDAV (CalDAV). RFC 4791 (Proposed Standard) (2007). URL http://www.ietf.org/rfc/rfc4791.txt. Updated by RFCs 5698, 6638, 6764

59. Deering, S., Hinden, R.: Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard) (1998). URL http://www.ietf.org/rfc/rfc2460.txt. Updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112

60. Diersk, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard) (2008). URL http://www.ietf.org/rfc/rfc5246.txt. Updated by RFCs 5746, 5878, 6176

61. dotCloud: ZeroRPC (2013). URL http://zerorpc.dotcloud.com/. Accessed 2014-02-19

62. Dubuisson, O., Fouquart, P.: ASN.1: Communication Between Heterogeneous Systems. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)

63. Dusseault, L.: HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV). RFC 4918 (Proposed Standard) (2007). URL http://www.ietf.org/rfc/rfc4918.txt. Updated by RFC 5689

64. ECMA International: Standard ECMA-262 – ECMAScript Language Specification (2011). URL http://www.ecma-international.org/publications/standards/Ecma-262.htm. Accessed 2014-02-21

65. Eddy, W.: TCP SYN Flooding Attacks and Common Mitigations. RFC 4987 (Informational) (2007). URL http://www.ietf.org/rfc/rfc4987.txt

66. Endres-Niggemeyer, B.: The mashup ecosystem. In: Semantic Mashups, pp. 1–51. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

67. Fette, I., Melnikov, A.: The WebSocket Protocol. RFC 6455 (Proposed Standard) (2011). URL http://www.ietf.org/rfc/rfc6455.txt

68. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard) (1999). URL http://www.ietf.org/rfc/rfc2616.txt. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585

69. Fielding, R., Lafon, Y., Reschke, J.: Hypertext Transfer Protocol (HTTP/1.1): Range Requests. RFC 7233 (Proposed Standard) (2014). URL http://www.ietf.org/rfc/rfc7233.txt

70. Fielding, R., Nottingham, M., Reschke, J.: HyperText Transfer Protocol (HTTP/1.1): Authentication. RFC 7235 (Proposed Standard) (2014). URL http://www.ietf.org/rfc/rfc7235.txt

71. Fielding, R., Reschke, J.: HyperText Transfer Protocol (HTTP/1.1): Authorization. RFC 7234 (Proposed Standard) (2014). URL http://www.ietf.org/rfc/rfc7234.txt

72. Fielding, R., Reschke, J.: HyperText Transfer Protocol (HTTP/1.1): Conditional Requests. RFC 7232 (Proposed Standard) (2014). URL http://www.ietf.org/rfc/rfc7232.txt

73. Fielding, R., Reschke, J.: HyperText Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230 (Proposed Standard) (2014). URL http://www.ietf.org/rfc/rfc7230.txt

74. Fielding, R., Reschke, J.: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231 (Proposed Standard) (2014). URL http://www.ietf.org/rfc/rfc7231.txt

75. Fielding, R.T.: REST: Architectural styles and the design of network-based software architectures. Phd thesis, University of California, Irvine (2000)

76. Fielding, R.T.: Rest apis must be hypertext-driven (2008). URL http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven. Accessed 2014-07-18

77. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Trans. Internet Technol. 2(2), 115–150 (2002)

78. Ford, A., Raiciu, C., Handley, M., Bonaventure, O.: TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824 (Experimental) (2013). URL http://www.ietf.org/rfc/rfc6824.txt

79. Ford, B.: Structured streams: A new transport abstraction. SIGCOMP Comput. Commun. Rev. 37(4), 361–372 (2007)

80. Forno, F., Saint-Andre, P.: XEP-0072: SOAP Over XMPP (2005). URL http://xmpp.org/extensions/xep-0072.html. Accessed 2014-07-25

81. Freed, N., Borenstein, N.: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045 (Draft Standard) (1996). URL http://www.ietf.org/rfc/rfc2045.txt. Updated by RFCs 2184, 2231, 5335, 6532

82. Freed, N., Borenstein, N.; Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. RFC 2046 (Draft Standard) (1996). URL http://www.ietf.org/rfc/rfc2046.txt. Updated by RFCs 2665, 7989, 5147, 6657

83. Freier, A., Karlton, P., Kocher, P.: The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic) (2011). URL http://www.ietf.org/rfc/rfc6101.txt

84. Furushashi, S.: MessagePack (2013). URL http://mmappack.org/. Accessed 2014-02-21

85. Galiegue, F., Zyp, K., Court, G.: JSON Schema: core definitions and terminology (2013). URL http://json-schema.org/doc/draft-zyp-json-schema/. Accessed 2014-07-18

86. Garnock-Jones, T.: Reverse HTTP (2010). URL http://reversehttp.org/reverse-http-spec.html. Accessed 2014-03-04

87. Garrett, J.J.: AJAX (2005). URL http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications. Accessed 2013-03-27

88. GlassFish Project: Open Message Queue (2014). URL https://glassfish.org/doc/draft-zyp-json-schema/. Accessed 2014-02-20

89. Google Developers: Google Cloud Pub/Sub (2014). URL https://developers.google.com/pubsub/overview. Accessed 2014-07-18

90. Google Developers: Google Cloud Pub/Sub (2014). URL https://developers.google.com/discovery/. Accessed 2014-07-18

91. Gottschalk, K., Graham, S., Kreger, H., Snell, J.: Introduction to web services architecture. IBM Systems Journal 41(2), 170–177 (2002)
241. Potti, P.K.: On the design of web services: Soap vs. rest. UNF
240. Postel, J., Reynolds, J.: File Transfer Protocol. RFC 959 (INTERNET STANDARD) (1981). URL http://www.ietf.org/rfc/rfc959.txt http://svn.cometd.org/code.google.com/p/pubsubhubbub/.
Accessed 2014-07-02
242. PrismTech: OpenSplice DDS Community (2014). URL http://www.prismtech.com/opensplice/opensplice-dds-community. Accessed 2014-07-24
243. ppushububhub: A simple, open, webhook based pushub protocol & open source reference implementation (2014). URL https://code.google.com/p/ppushububhub/. Accessed 2014-07-02
244. Rackspace: Cloud Queues (2014). URL http://docs.rackspace.com/queues/api/v1.0/cq-gettingstarted/content/DB_Overview.html. Accessed 2014-07-21
245. Rady, M.: Formal definition of service availability in cloud computing using owl. In: Computer Aided Systems Theory - EUROCAST 2013, Lecture Notes in Computer Science, vol. 8111, pp. 189–194. Springer Berlin Heidelberg (2013)
246. Rady, M.: Generating an excerpt of a service level agreement from a formal definition of non-functional aspects using owl (2014). Conceptual Modelling with Specific focus on Service-Oriented Systems, Special Issue of the Journal of Universal Computer Science (to appear)
247. Ramsdell, B., Turner, S.: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification. RFC 5751 (Proposed Standard) (2010). URL http://www.ietf.org/rfc/rfc5751.txt
248. Red Hat Enterprise: MRG – Messaging, Realtime, Grid (2009). URL http://www.redhat.com/t/pdf/MRG_brochure_web.pdf. Accessed 2014-07-21
249. Rescorla, E.: HTTP Over TLS. RFC 2818 (Informational) (2000). URL http://www.ietf.org/rfc/rfc2818.txt. Updated by RFCs 5785, 7230
250. Rescorla, E.: SSL and TLS: designing and building secure systems. Addison-Wesley (2001)
251. Rescorla, E., Modadugu, N.: Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard) (2012). URL http://www.ietf.org/rfc/rfc6347.txt
252. Restlet: The Leading Web API Platform for Java (2014). URL http://restlet.org/. Accessed 2014-05-05
253. Ristic, D.: WebRTC data channels (2014). URL http://www.httpsrocks.com/en/tutorials/webrtc/datachannels/. Accessed 2014-06-05
254. Roskind, J.: QUIC: Multiplexed stream transport over udp (2013). URL https://docs.google.com/document/d/1RJllIgx_VeKWyW6Lr8SZ-aqagQe7rFV-ev2jRFuoVD34. Accessed 2014-04-30
255. RSS Advisory Board: RSS 2.0 Specification (2009). URL http://www.rssboard.org/rss-specification. Accessed 2014-02-21
256. RTI: Connext DDS Software (2014). URL http://www.rti.com/products/index.html. Accessed 2014-07-24
257. Russell, A.: Comet: Low latency data for the browser (2006). URL http://infrequently.org/2006/03/comet-lowlatency-data-for-the-browser/. Accessed 2014-02-20
258. Russell, A., Wilkins, G., Davis, D., Nesbitt, M.: Bayeux Protocol – Bayeux 1.0.0 (2007). URL http://svn.cometd.org/trunk/bayeux/bayeux.html. Accessed 2014-03-04
259. Ryck, P., Decat, M., Desmet, L., Piessens, F., Joosen, W.: Security of web multiplexes: A survey. In: Information Security Technology for Applications, Lecture Notes in Computer Science, vol. 7127, pp. 223–238. Springer Berlin Heidelberg (2012)
260. Saint-Andre, P.: Extensible Messaging and Presence Protocol (XMPP): Address Format. RFC 6122 (Proposed Standard) (2011). URL http://www.ietf.org/rfc/rfc6122.txt
261. Saint-Andre, P.: Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Proposed Standard) (2011). URL http://www.ietf.org/rfc/rfc6120.txt

Certificate_Pinning Slides.pdf. BlackHat USA. Accessed 2014-07-29
222. OW2 Consortium: JORMA: Java Open Reliable Asynchronous Messaging (2014). URL http://joram.ow2.org/technical.html. Accessed 2014-07-23
223. OWASP: Certificate and Public Key Pinning (2014). URL https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning. Accessed 2014-07-29
224. Paoli, J.: Speed and mobility: An approach for http 2.0 to make mobile apps and the web faster (2012). URL http://blogs.msdn.com/b/interopability/archive/2012/03/25/speed-and-mobility-an-approach-for-http-2-0-to-make-mobile-apps-and-the-web-faster.aspx. Accessed 2014-06-04
225. Paterson, I., Saint-Andre, P., Stout, L., Tilianus, W.: XEP-0206: XMPP Over BOSH (2014). URL http://xmpp.org/extensions/xep-0206.html. Accessed 2014-07-23
226. Paterson, I., Smith, D., Saint-Andre, P., Moffitt, J.: XEP-0124: Bidirectional streams Over Synchronous HTTP (BOSH) (2010). URL http://xmpp.org/extensions/xep-0124.html. Accessed 2014-03-04
227. Pautasso, C.: Composing restful services with jopera. In: Software Composition, Lecture Notes in Computer Science, vol. 5634, pp. 142–159. Springer Berlin Heidelberg (2009)
228. Pautasso, C.: RESTful Web Service composition with BPEL for REST. Data & Knowledge Engineering 68(8), 851–866 (2009)
229. Pautasso, C.: Bpmn for rest. In: Business Process Model and Notation. Lecture Notes in Business Information Processing, vol. 95, pp. 74–87. Springer Berlin Heidelberg (2011)
230. Pautasso, C., Wilde, E.: Why is the web loosely coupled?: A multi-faceted metric for service design. In: Proceedings of the 18th International Conference on World Wide Web, WWW’09, pp. 911–920. ACM (2009)
231. Pautasso, C., Wilde, E.: Restful web services: Principles, patterns, emerging technologies. In: Proceedings of the 19th International Conference on World Wide Web, WWW’10, pp. 1359–1360. ACM (2010)
232. Pautasso, C., Zimmermann, O., Leymann, F.: Restful web services vs. “big” web services: Making the right architectural decision. In: Proceedings of the 17th International Conference on World Wide Web, WWW’08, pp. 805–814. ACM (2008)
233. PeerCDN: FAQ (2014). URL https://peercdn.com/faq.html. Accessed 2014-07-27
234. Phelan, T.: Datagram Transport Layer Security (DTLS) over the Datagram Congestion Control Protocol (DCCP). RFC 5238 (Proposed Standard) (2008). URL http://www.ietf.org/rfc/rfc5238.txt
235. Pivotal Software, Inc.: AMQP URI Specification (2014). URL http://www.rabbitmq.com/uri-spec.html. Accessed 2014-07-27
236. Pivotal Software, Inc.: RabbitMQ (2014). URL https://www.rabbitmq.com/. Accessed 2014-02-19
237. Postel, J.: User Datagram Protocol. RFC 768 (INTERNET STANDARD) (1980). URL http://www.ietf.org/rfc/rfc768.txt
238. Postel, J.: Internet Protocol. RFC 791 (INTERNET STANDARD) (1981). URL http://www.ietf.org/rfc/rfc791.txt. Updated by RFCs 1349, 2474, 6864
239. Postel, J.: Transmission Control Protocol. RFC 793 (INTERNET STANDARD) (1981). URL http://www.ietf.org/rfc/rfc793.txt. Updated by RFCs 1122, 3168, 6093, 6528
240. Postel, J., Reynolds, J.: File Transfer Protocol. RFC 959 (INTERNET STANDARD) (1985). URL http://www.ietf.org/rfc/rfc959.txt. Updated by RFCs 2228, 2640, 2773, 3659, 5797, 7151
241. Potti, P.K.: On the design of web services: Soap vs. rest. UNF Theses and Dissertations, Paper 138 (2011). URL http://digitalcommons.unf.edu/etd/138/
