Practical issues in protocol design

Sami Vaarala, 2006

Introduction

Protocol design can be approached from a (semi-)formal viewpoint, using e.g. state machine models and other such tools. It can also be approached from a more practical viewpoint. In what follows I'll present some practical viewpoints on protocol design. My experience comes from a career in IP security and mobility protocols, during which I've been involved in implementation work on standard protocols (such as IPsec, Mobile IPv4, RADIUS, L2TP, PPP, TLS, etc), as well as several company proprietary protocols for device configuration management and monitoring.

From a system viewpoint protocols can be seen as black boxes. They provide well-defined and useful upper layer services, by using lower layer services to actually implement the services. Protocols should promote interoperability of various implementations, and be future-proof, i.e. maintainable and extensible.

But protocols can also be viewed from a software viewpoint. Whenever two software modules exchange data over a network, they always follow some set of rules, as defined by their program code. These rules, whether they have been defined in a document or not, constitute a protocol. So, protocols happen whether we want them to or not. From this viewpoint, our goal is to ensure that such software interactions are as well-defined as possible, and that implementors can come to agreement when implementing such software.

A pragmatic definition for a protocol is thus the set of rules which software modules use to exchange data over a network (typically), with the hope of accomplishing some useful task. Instead of defining the rules as code, we hope to step up one level and describe a sort of "meta-code", which applies to multiple implementations. Of course we hope that this meta-code – the protocol specification – is shorter and easier to understand than actual code, and encompasses multiple implementations. The set of rules which implementations should follow are described in a protocol specification in a compact and implementation-independent manner.

Note that there are alternatives to describing protocols in specification documents. For instance, we might just produce a simplified, low performance reference implementation which everyone can compare their implementation to. Often, however, it is hard to ensure that the reference implementation actually works in a useful way, unless its functionality is boiled down into a simplified form – the protocol. Sometimes previously deployed implementations may become the actual reference implementation. This is, understandably, sometimes a rather messy situation one would like to avoid.

Protocol specifications are often understood to be public, or at least shared by multiple companies. This is not always the case: proprietary protocols are often necessary because standard protocols simply don't cover every aspect of real product needs. For instance, to make user experience as smooth as possible, some proprietary protocols may be needed to "bootstrap" information required by standard protocols. Also, if an application is especially novel, one may not find applicable standard protocols. Careful protocol design

and specification is a bonus for proprietary protocols as well, as a company may need to maintain compatibility with the protocols for multiple products versions in the future.

But how does one actually design a protocol? As in all design, we first need to define the goals / requirements that we're designing the protocol to. Second, we need the actual design, which is written into the form of a specification. Third, we attract people to implement the specification, and try to make the implementations interoperate with one another. This provides feedback to the specification, hopefully making it as universally understandable as possible so as to minimize implementation mistakes.

We'll look at some aspects of these steps and shortly discuss IETF standardization.

Goal setting

Protocol design as multi-goal optimization

Protocols have conflicting goals, e.g. simplicity, performance, functionality, security. Protocol design is thus necessarily multi-goal optimization, with multiple goals and multiple measures of success. So called Pareto-optimal solutions are in some sense optimal: one aspect of the protocol cannot be improved without a negative effect on the others. Good protocols are close to Pareto-optimal points, while bad protocols are not. For instance, a bad protocol may be both slow and complex, which is a bad tradeoff.

Multi-goal optimization requires prioritization of goals. Having a clear view of the relative priorities of goals (e.g. do we prefer simplicity over performance?) ensures that design trade-offs are made consistently throughout the protocol. From an abstract viewpoint, prioritized goals allow us to choose between Pareto-optimal points.

Some high-level goals

Functionality

- Clearly defined minimum functionality and “use cases”.
- Completeness of functionality within desired scope.

Security

- Realistic threat model, which allows us to reason about the security of the system. The threat model identifies assets, threats, and assumed attacker capabilities.
- Security assumptions (which security issues are out of protocol scope) and security objectives.

Performance, scaling, and reliability

- All protocol implementations have a need for performance, but performance goals for protocols are difficult to define exactly. Rather, a protocol must not, because of its design, impede performance.
- A goal for supported “transactions”, “connections”, “sessions”, etc. is useful, even if on an order-of-magnitude ballpark (e.g. typical server should be able to handle 1M transactions/hour).

Compatibility

- Compatibility with previous and future protocol versions is usually important.
- Compatibility with fielded implementations, which may be buggy and noncompliant, is also often important.
- "Use case compatibility" with another protocol is sometimes an appropriate model, e.g. a drop-in replacement for an old protocol (IKEv1 => IKEv2).

Implementation restrictions

- Even through protocols are not tied to software as such, we must still take it into account.
- Intended implementation model is a rough idea of how the protocol would be typically implemented: userspace process (TLS), kernel driver (IPsec), etc.
- Minimum memory, CPU, and network performance assumptions guide protocol design. E.g. if protocol implementations are assumed to have very little memory, state (buffering, message queues, etc) must be minimized.

Related issues

Optimality vs. complexity trade-off. Any protocol can be improved marginally (e.g. performance or functionality) by adding complexity. Minimal complexity is a universal design goal because it makes correct implementation more likely. We must thus accept reasonable non-optimality in protocols in order to get simplicity. What constitutes "reasonable" depends on the protocol usage.

Design from scratch or reuse an existing protocol? If a protocol needs to be designed from scratch, one should have a good understanding why. Existing protocols are preferable if they are applicable, perhaps with universal or vendor-specific extensions, because others are already familiar with them and there may be an existing code base. Also, even if the main protocol is designed from scratch, sub-protocols can often be reused – e.g. use TLS or IPsec for traffic protection or RADIUS for authentication. Cryptographic protocols are extremely difficult to design – it takes years from even the most qualified people – and should almost never be designed from scratch.

Protocol "exposure" to other parties. Sometimes protocols are internal to a product and do not need to be exposed to others (e.g. a back-plane clustering protocol). Sometimes protocols may form a multi-vendor platform, so protocol audience is much larger. It is useful to distinguish with critical protocols which are used for decades, and transition or support protocols which are used for years but phased out or replaced on the medium term.

Standardization goal may range from a closed proprietary protocol, to a de facto or official standard. Getting a protocol accepted by a standards body, such as IETF or 3GPP, may take years. Often vendors settle for de facto standards, where the specification is public, multiple vendors implement the specification, but no official standardization status exists.

Characteristics of a good protocol

Clear purpose and reason for existence. Protocol scope is coherent: there is little

overlap with other protocols and the protocol generalizes to a wide variety of applications. The protocol lifetime is well understood (1-year transition vs. 10-20 years).

Appropriate protocol mechanisms. Protocol contains complete and coherent mechanisms for desired functionality, with minimal duplicate mechanisms. Maximum functionality is provided for minimal complexity. There is a clean path for extensibility and version compatibility.

Specification documents follow 3C principle: Completeness, Consistency, Clarity.

Implementations and implementors. There are multiple interested parties writing protocol implementations, and there is a reference implementation. The protocol allows product and version identification for testing and eventual workarounds.

Threat model well understood. Threats are realistic, defined explicitly, and the countermeasures in the protocol are appropriate for the threats (not too weak but not too strong either).

Reuse existing protocols wherever possible. The protocol reuses existing protocols wherever possible (e.g. Extensible Autentication Protocol, EAP, for user authentication). This modularizes and minimizes code base.

Allows various implementation models. The protocol can be implemented as a simple subset of full functionality and still work reasonably well; the protocol also allows a fully fledged, highly optimized and scalable implementation. If the protocol has a "front-end" and a "back-end", the "front-end" interface is minimal but "back-end" protocols may be arbitrary.

Allow high availability (HA) and load balancing (LB). The protocol may cover HA and LB directly, but if it doesn't, it should not inhibit the use of out-of-scope HA/LB solutions. This is important because any large scale protocol needs to eventually deal with scaling up the solution.

Graceful extensibility and degradation. It should be possible to experiment with proprietary and other extensions without risking main protocol functionality; newer protocol versions should also be able to coexist with older ones. If there is no agreement on extensions, the protocol should gracefully degrade to a base standard where some base functionality is still available.

Clearly cut implementation requirements. The protocol specification clearly defines requirements which are mandatory for interoperability. Implementations fulfilling mandatory requirements should have some guarantees of interoperability. Non-essential features should not be defined as mandatory.

Protocol diagnostics. The protocol should support easy diagnostics by administrators. Eventually most protocols are diagnosed by administrators, so allowing verbose errors to be passed in protocol messages in a readable form is useful. E.g. HTTP responds with a generic error class (4xx), refinements (404), and an error string; the generic error class and refinements control software actions, while error string is for administrators and users.

IPR issues. Ideally the protocol has no Intellectual Property Rights (IPR) dependencies, particularly patents which need to be licensed in order to use the protocol. A less favorable situation is where interoperability is possible without IPR licensing, but

optimized implementation may not be. In the worst case any practical implementation requires some form of licensing.

Protocol design issues

The three major issues

The three major issues covered by protocol design are:

(1) protocol behavior

(2) protocol syntax (data)

(3) implementation guidance.

Protocol behavior describes how protocol processing works. This is often done by describing protocol states, state transitions, messages, and timers; and through descriptions of successful protocol runs and use cases. Protocol specification should also describe how (expected) errors are to be handled.

Protocol syntax, or "bits-on-the-wire", describes the bits and bytes used by the protocol – the Protocol Data Units (PDUs). Although syntax definition is usually relatively simple (at least in comparison to good description of behavior), there are still common mistakes in many deployed protocols.

Implementation guidance is added to specifications to increase the probability of successful implementation. Such guidance may be informative (i.e. place no new requirements) or normative (i.e. place new requirements which must be followed). Guidance can be given through warnings against certain bad implementation practices, as suggestions for good implementation approaches, test vectors, example protocol runs, etc. Implementation guidance is especially important for security features, for instance, stating when an implementation must remain stateless to avoid denial-of-service vulnerability.

Conceptual design

Conceptual design is a high level design where simplified use cases of the protocols are used to model the protocol. The goal is to prevent major design mistakes, to define protocol entities and protocol surroundings (e.g. other protocols, users), and to select the basic interaction model (request-response, peer-to-peer, etc) as well as the basic transactions/operations that the protocol should provide.

Typical protocol session life-cycle

Typical protocols handle *sessions*, which have state (i.e. dynamic variables related to the session); for instance, an SMTP session allows sending of e-mail. Session management includes (1) session establishment, (2) session maintenance and "steady state", and (3) session teardown.

The session establishment, or handshake, covers issues such as identification of communicating parties (identity, vendor and product version), protocol capability negotiation, and protocol version negotiation. Parties are often authenticated during session establishment, and some protocols offer a session resumption function which allows an earlier session to be resumed quickly without cumbersome session establishment (cf. TLS protocol session resume). The negotiation phase is important

because it plays a major role in extending the protocol gracefully in later versions.

A typical design issue in negotiation of features, capabilities, and options is how to represent the alternatives. Two basic approaches are used: (1) "a la carte", where every option has a separate list of acceptable values, and the selected value for each option is independent of other options; and (2) "suites", where a single integer (or other identifier) identifies the values for all options. The advantage of the former is that it allows complete control, although it may result in complex implementation and difficulties in interoperating and agreeing upon suitable combinations of options. The advantage of the latter is that protocol designers can come up with combinations of options which make sense, but may suffer from a combinatoric explosion, where the amount of suites become very large very quick. The IKEv1 key management protocol uses the a la carte approach, while the TLS encryption protocol uses suites.

Negotiation often follows an offer-response sequence, where one party, often the initiator of the session, offers acceptable values while the responder selects particular values for the session. Some protocols have more complex negotiation sequences; for instance, the Point-to-Point Protocol (PPP) can use several roundtrips before options are agreed upon (five or more roundtrips not being uncommon).

Session maintenance and steady state processing is of course the centerpiece of a protocol. Typical issues in defining steady state processing is understanding performance concerns and selecting protocol primitives or transactions accordingly. Good data design is also important.

Session teardown is usually quite straightforward. It is desirable to shut down cleanly, so that both peers end up in a known state. However, because networks cannot be typically fully trusted, protocols and implementations must support unclean teardown anyway.

Data formats

Conceptual design should give you an idea of what sort of data items need to be represented in the protocol. For instance, the SMTP protocol has commands with parameters, responses, and e-mail data. E-mail data, in turn, has headers and content, with headers usually represented by a header name and a header value. At this stage it is also important to know which data is visible to the user, and might thus benefit from localization or at least a wide (ISO-10646 or Unicode) character set.

The next step is to select the bits and bytes to represent the conceptual data items. Although this choice is relatively trivial, there are multiple considerations:

- Intuitiveness is important so that implementors and administrators can easily implement and debug the protocol. This improves the probability of correct implementation. Implementors should be able to remember the basic syntax intuitively without constant reference. This is achieved e.g. by consistent design choices throughout the protocol.
- Canonical or non-canonical format. A canonical format has only one representation for a specific conceptual data item. For instance, an attribute-value pair has only one binary encoding. This is convenient because it limits variance across protocol implementations and works well with e.g. digital signatures. Unfortunately many real-world protocols don't have canonical encoding – IKEv1, for instance, does not, which leads to interoperability

problems where attribute-value pairs encoded using a 3-byte data value (instead of a 4-byte data value) cause problems in some implementations.

- Reuse of existing formats: previous protocols or previous protocol version, XML, ASN.1, ...
- Binary vs. textual protocol: is protocol data relatively straightforward, for a human, to read? Textual format has almost always a performance impact in both data size and processing complexity, but is often easier to implement and diagnose in the long run. Performance can be improved by e.g. data compression. If data complexity of the protocol is low, textual formatting may have little value.
- Human readable vs. machine readable strings. ISO 10646 / Unicode and UTF-8 encoding is often preferred, because it represents 7-bit ASCII characters in an ASCII-compatible manner, while allowing a wide variety of languages to be used. It is important to categorize human readable strings into truly end user strings (which are shown to even non-technical users) and administrator/developer strings (which are used for development and debugging, consider e.g. error details in protocol). Localization of strings imposes very stiff requirements and should not be supported unless a real need exists.

Extensible Markup Language (XML) may be a good default choice for several reasons: (1) it is probably the best widely accepted standard for formatting of structured data; (2) multiple programming languages support XML parsing, Schemas, XML Stylesheet Transformations (XSLT), and other standards; (3) XML is extensible through attributes, elements, and name spacing; (4) XML is human readable for administrators and programmers. However, there are also disadvantages, such as: (1) non-optimal encoding which requires compression or alternative binary encoding for optimized size; (2) data model is sometimes confusing, e.g. whether to use attributes or element content; (3) XML DTD and namespace interactions are somewhat messy; and (4) processing of large amounts of XML can have a significant performance impact compared to processing binary data.

Abstract Syntax Notation One (ASN.1) is also used in several protocols, especially cryptographic ones. In principle ASN.1 is OK, but is a bit painful in practice for many reasons. It is also not human readable, but separate tools are required to inspect contents properly. It is, however, quite compact.

A typical binary format used in protocols is the "Header + Attribute-Value-Pairs (AVPs)" model; it used by most extensible IP protocols such as IKE, Mobile IP, RADIUS, Diameter, etc. Although encoding attribute-value pairs is in principle simple – attribute type and attribute value are encoded in some fashion – there is unfortunate variance between protocols. Some protocols support the notion of critical vs. non-critical AVPs: a receiver which does not recognize a particular critical AVP is required to abort while an unrecognized non-critical AVP is skipped. Some protocols support vendor-specific AVPs, e.g. by prefixing attribute type with a vendor identifier (IANA enterprise number or other). Maximum size and of types and values also varies, as well as whether a canonical encoding is used (e.g. use shortest encoding always). IP protocols also often strive for byte alignment, so that each data element is aligned to a multiple of its size – in practice, IP protocol headers are usually aligned to 4 bytes.

High availability (HA) and load balancing (LB)

Most successful protocols are eventually deployed in environments where service availability and scalability are important. Service availability means that a user who desires to use the service, gets the service when needed. Scalability means that the amount of users, sessions, transactions, or other important parameters can be increased without a severe bottleneck such as performance of a single computer, a single network element, etc. These are typical requirements in operator environments: money is lost when service is not available, while a poorly scalable service requires expensive workarounds.

The term high availability (HA) is often used to refer to the goal of high service availability. It is typically implemented through server clustering, redundant ISP connections, and other redundancies in design. Load balancing (LB) is a particular technique related to scalability; load balancing allows multiple servers to share session load so that the total load can be balanced in a (semi-)optimal manner between the available servers.

High availability and scalability have two main relationships two protocols:

1. Inside scope – the protocol defines high availability and scalability mechanisms directly.
2. Outside scope – the protocol does not define such mechanisms but allows an efficient high availability and scalable implementation "in the backend"

From the outset it might seem that providing HA & LB is a product development issue. This is true for the most part, but protocol specification may inhibit high availability or load balancing implementation or make such an implementation very difficult. For instance, the IPsec security model requires tight synchronization of state among servers in a server cluster, thus making IPsec clustering difficult to implement correctly.

Ideally, multiple servers sharing user load should be from multiple vendors to increase software diversity and minimize the chance of common failure modes. This would require a protocol with a standardized clustering mechanism or some standardized out-of-band clustering protocol. Unfortunately, multi-vendor clustering is often infeasible.

A protocol designer should have a conceptual view of how high availability and scalability are achieved. For instance, when a server fails, are sessions lost and reinitiated or are sessions somehow transferred to another server? There are two basic models for

high availability and scalability:

- Client-driven, where the client detects lack of service and reinitiates the session. Often client-driven approaches use DNS (in a round robin manner) or some other redirection protocol to divert user load. DNS is nice because it is a standard and used in most cases anyway, but has caching issues.
- Server-driven, where the server implements session state replication and transparent server failover without client knowledge. Often uses a virtual IP address shared by multiple servers.

Other design issues

Dealing with overload. When server load increases, performance can either reduce gracefully, or it can completely melt down so that almost no user gets service. For instance, if a server carelessly accepts all incoming connections, it will soon be unable to maintain sessions with any users, and the sessions will eventually time out leaving all users out of service.

A good protocol should be implementable so that performance undergoes graceful degradation when load increases. Ensuring graceful degradation requires, among other things, good timer handling, some control over how new sessions are accepted or rejected, and perhaps an explicit "busy error" mechanism which allows a server to fend off overload. Typically only implementation guidance is given, as it is usually sufficient.

Security. Because designing security protocols, especially cryptographic protocols, is so difficult, it is almost always a good idea to just use an existing secure transport protocol for your data. Usually this means (a) IPsec for packet-based (layer 3) data, (b) SSL/TLS for stream (layer 4) data, or (c) PGP (or other similar alternatives) for application (layer 7) data objects.

It is important to remember that security protocols are designed to resist some, but not all, security threats. For instance, most protocols do *not* attempt to resist denial-of-service, which is often difficult because just the use of TCP as a transport, for instance, may make denial-of-service resistance unachievable. Careful analysis of actual relevance of denial-of-service threats should be done.

In addition, there are always application specific threats which cannot be addressed by the security protocol and need to be considered application-by-application. Security protocols also often need some supporting out-of-band mechanisms, such as how to exchange certificates securely.

User authentication is a complex topic. Organizations have an extremely wide variety of authentication infrastructures, particularly operators and large enterprises, and they usually need to make new protocols interoperate with the old infrastructure to some extent.

There are some existing protocols to deal with this. Extensible Authentication Protocol (EAP) contains a number of authentication methods to deal with "legacy", or non-PKI, user authentication. RADIUS and Diameter protocols are convenient for interfacing with existing infrastructure, e.g. proprietary user databases.

Note that authentication models cannot be fully abstracted away – they do have a protocol impact. For instance, some authentication methods, such as RSA SecurID authentication,

(sometimes) require multiple roundtrips, whereas plain password authentication does not. The protocol needs to accommodate the required interactions.

Network Address Translators (NATs). NATs are fully integrated into modern IPv4 networks and will, unfortunately, probably be relevant for IPv6 networks as well. It is futile to try to resist NATs – any real world protocol implementation must function through NATs somehow. "NAT traversal" is typically done differently for each protocol (e.g. IPsec and Mobile IPv4), but there is some common work, such as the STUN protocol (RFC 3489).

Leniency and strictness. A common IETF design guidance is to be conservative in what you send and liberal in what you accept. The intent is to foster interoperability between implementations of varying quality. However, when new protocols are designed, strictness may be sometimes more desirable: differing implementations complicate development, testing, and administration. When a new protocol is defined, strictness is more feasible than when interoperating with an old protocol. Especially security protocols should be strict.

Standardization at the IETF

IETF in brief

Internet Engineering Task Force (IETF) develops Internet protocols which are published as RFC documents. Internet Architecture Board (IAB) and Internet Engineering Steering Group (IESG) guide and control IETF activity.

IETF is organized into areas and working groups within areas. There are currently eight areas – applications, general, internet, operations and management, real-time applications and infrastructure, routing, security, and transport – with dozens of working groups in most areas. Each working group (WG) focuses on a concrete goal, defined by its charter, and produces Internet-Drafts which later mature into RFCs. The focus is very practical – get the problem solved one way or the other.

IETF has open participation, and there is no membership as such. Anyone can comment on WG mailing list, and documents are public and free of charge.

How standards are created

Basic components of IETF standardization include:

- Working group formation – idea for standardization is discussed in a Birds-of-a-Feather (BOF) meeting, and the working group is formed with a specific charter.
- The charter describes working group scope and goals – what sort of services the protocols defined by the WG should provide, and what other outputs and activities the WG should have. Charter is the backbone for making judgement calls about what activities belong in the WG; it is reviewed from time to time to accommodate changes in circumstances.
- The working group process is relatively simple – working group meetings take place during IETF meeting three times a year. Mailing lists are used for the actual work of the WG, as some participants may not be able to make the

physical meetings. Decisions are made using a rough consensus principle, where the WG chairs determine whether there is a rough consensus on an issue. Sometimes the consensus is indeed rough, but progress is hopefully quicker than with a more formal decision process.

- WG creates Internet-Drafts (I-Ds) which are versioned work-in-progress document with no official status. They are circulated within the WG (and other IETF participants) for comments and editing.
- Standards are completed by first issuing a working group last call (LC), which indicates the final chance for WG comments on the document. The document then undergoes wider review and proceeds to an IETF last call. Finally, once all last call comments have been addressed, the document undergoes final editing and publication as an RFC. RFCs do not change after publication – not even to fix bugs in the specification. A newer RFC may obsolete an older one, fixing problems in the previous RFC.

Politics in and out of IETF

Protocol standardization is a political process – they are large processes with multiple stakeholders who have conflicting goals. There is thus, inevitably, some "power struggle" to steer decisions. This is not inherently good or bad – there are both positive and negative impacts of political processes.

Negative impacts include:

- "Committee design", where multiple overlapping features and options are added to satisfy participants and "save face". Often such features are specified but never used in real life, and unnecessarily complicate a protocol.
- De facto off-shoots, where disagreeing groups within the organization go for a "de facto" protocol or protocol extensions to solve a particular issue. Such extensions may become important in practice if no standard solution exists, again complicating the overall protocol. For instance, IPsec legacy (non-PKI) authentication is somewhat of a mess because there are multiple de facto solutions as well as some semi-standard ones.
- "IPR submariners". Submarine IPR (Intellectual Property Rights, typically patents) means waiting for a standard to be released and then claim you have IPR on the standard. Although most standardization bodies try to avoid IPR problems, it is extremely difficult.
- Participating in the standardization process may range from being very easy to being next to impossible, despite apparent openness (of e.g. IETF). The political barrier of the working group influences the difficulty – e.g., if there are many stakeholders with great interest in not doing some activity, they usually have a large influence in their favor.

Overall politics may thus result in a technically non-optimal protocol, to satisfy some process limitations.

However, politics do also have positive effects. For instance, processes which end up in endless arguments and deadlock can often be resolved one way or the other if a suitably powerful group enforces a decision. Indeed, it has been argued that politics are necessary

to drive any large scale process, because technical objectivity is simply not achievable in practice.