Protocol Design Assumptions and PEPs

Jörg Ott
3 May 2019

Introduction

Protocol designers have often made – implicit or explicit – assumptions about the deployment environments of their protocols. These assumptions may be about numerous system or network characteristics, including the available data rate, the packet loss, path symmetry, path stability (or time-scale of path or connectivity changes), network transparency, and end-to-end latency. Such assumptions may have been made explicit – when protocols are specifically designed for LANs – or may have been simply implicit by designing for an Internet of the "today" back then, e.g., an Internet without many wireless or satellite networks or without much mobility; or local networks before data centre scale capacity and RTTs.

End-to-end latency (along with short-term disconnections) has, besides data rate, probably had the most profound impact on protocol performance and given rise to substantial and quite diverse effort in protocol optimization mechanisms "in the middle". These intermediaries (or: middleboxes), also dubbed Performance Enhancing Proxies (PEPs) or WAN optimizers, provide different flavors of performance optimizations at different layers [RFC 3135], quite often focusing on transport but also reaching into application protocols. Such intermediaries arguably have notable impact on successfully expanding the deployability of these protocols in setups beyond their original target environments. But they also yield a number of – known – unpleasant side effects:

- They may break the end-to-end model of reliability, application logic, and security.
- They may contribute to the ossification of the Internet as they affect independent innovation at the edges because endpoints and intermediaries need to evolve in sync.
- They may generate the impression with ISPs and operators that certain performance levels are (perceived to be) only achievable if intermediaries can be deployed and thus create a demand on future protocols to be intermediary-friendly.

In this brief position paper, we look at one ancient and one timeless protocol that have traditionally suffered from poor performance when run natively outside their originally intended environment, investigate protocol abstractions (and APIs) for some of those protocols, and explore the lessons learned from those.

Two Case Studies

Server Message Block (SMB)

Our first case study covers a protocol for accessing resources across a local area networks, SMB that represents a protocol design expecting very low latency and sufficient network capacity for its operation – at least SMB1 did (later versions of SMB fixed the issues mentioned below). Protocols of similar nature include the X protocol (used by the X Window System) and, to a lesser extent, H.323 originally designed for extending ISDN-based multimedia conferencing into local IP-based packet networks.

SMB1 offers a large number of fine-grained protocol interactions (commands and subcommands) to access and interact with a remote machine – rather than providing a semantically richer abstraction via an API. Since the use and composition of these (sub)commands was entirely left up to the application designer, with very little apparently done in terms of functional aggregation inside the local stack, the application programmer had full control over how she codes the interaction of a local client and a remote file system. SMB performed reasonably well when run in a LAN environment, but performance degraded quickly when RTT increases. For example, we measured this effect for satellite links and found that performance decreased notably with growing latency: while copying a 15MB file over a LAN with RTT=1ms took (back then) 2.5s, the same operation took 6min to complete across a satellite link of equal capacity but with RTT=1s. An in-depth investigation revealed many unnecessary protocol exchanges: the exact same interactions were repeated many times, suggesting that the the RPC-style operations were used similar to a system-local procedure call: rather than caching information locally, the server was inquired over and over again. Indeed, when implementing an SMB PEP that understands the protocol logic and can provide caching to avoid round-trips to the remote server we found that these measures could improve performance substantially.

We find three main contributors to poor performance outside the originally envisioned environment: 1) the fine-granular operations and the resulting chattiness of the protocol, 2) the lock-step-style operation of the protocol and the lack of pipelining, and 3) the exposure of many functions at the API (along with its local implementation) that would allow for poor use of the protocol.

HTTP/1.1

Our second short case studies deals with a timeless classic, HTTP/1.1, broadly used throughout the Internet. HTTP/1.1 represents a class of transaction-based protocols that usually require a series of transactions to complete a simple semantically meaningful operation (such as “get this web page”). Similar protocols include SMTP, POP3, and IMAP4, among others.

HTTP provides transaction-based retrieval (we focus on its main use and leave the other HTTP methods aside for now) of resources that are combined in the HTTP client to form – in its main use – a complete view of a web page with numerous embedded objects. These resources may originate from the same or different sites and, for the former, may be retrieved (pipelined) via one or more transport connections. Its transaction-based nature makes HTTP a chatty protocol that requires potentially many interactions with a server to retrieve all sought content. HTTP inherently supports proxies and caches to reduce content retrieval latency (one metric being page load time). But using classic web caches has lost relevance because of a personalized web pages and less cacheable content and with the shift from unencrypted content retrieval to the use of TLS for encryption and server authentication in recent years. Caches have also shown a poor interaction with HTTP-based adaptive (HAS, DASH).

Not-so-much-standard intermediaries (PEPs, and others) have maintained their relevance for HTTP; often inserted transparently between clients and servers, they perform content adaptation, caching, compression, and possibly prefetching, the latter especially in networks where otherwise high RTTs would degrade user experience. (PEPs may also translate TCP to other transport flavors that cope better with, e.g., long fat pipes.) These techniques were shown to address the issue of HTTP chattiness very well, as long as web page design allows effective prefetching to happen. In contrast, server push as introduced by HTTP/2 seems to have seen only limited uptake. To cope with HTTPS, ISP/operator-deployed PEPs may gradually morph into the outer edges of CDNs or find other way to – “legitimately” – terminate TLS connections.

We find two main contributors to performance issues remedied by intermediary deployments: 1) the transaction-based nature paired with 2) the substantial complexity of web site design or, more generally, the use of HTTP as underlying protocol. (Server push could help with aspect 2) in some cases but not in all and doesn’t seem to have seen much deployment.) Deploying even the intended intermediaries is made difficult 3) by the unhappy design to conflate transport layer security (client to server or client to intermediary and intermediary to server) and application layer protection (end-to-end client to server) in a system where the transport layer ends may, but need not align with the application layer ends.

Lessons learned

While the two case studies briefly introduced above deal with quite different protocols (or protocol classes), we find two commonalities: 1) Chattiness in protocol design makes it hard to maintain efficiency across a range of environments (and they also create many state transitions in the respective implementations. Pipelining (instead of lock-step operation) can help in some cases by reducing the number of RTTs to wait but the underlying state machines still need to advance in sync. 2) A fine-grained API offers lots of flexibility but certain uses of the protocol “API” (or: service interface) may lead to poor performance that is not inherent to the protocol, but maybe also not obvious to the application using it. HTTP has been (mis)used as poor transport for many purposes, often sufficiently well engineered (with lots of effort) to make it

work. The expansion of CDNs (and thus the consolidation efforts) seek to transform the Internet back to emulate the well-performing smaller scale network, at least for the services using the CDN, while the protocol design and use may put those not on a CDN at a competitive disadvantage.

While we didn't discuss security in the first use case, entangling transport and application layer security is a third common issue in spite of which protocols work: 3) When application layer state is tied to a (secure) transport connection, e.g., first performing user authentication and then the subsequent actions, the cost (time, bits sent) of reestablishing a connection and resynchronizing state grows, impacting protocol robustness in the presence of short-term failures, e.g., due to mobility. Moreover, if protocols seek to support intermediaries, they ought to provide proper independent end-to-end application layer security support. The Session Initiation Protocol (SIP, RFC 3261 and many others) exhibit a design that, while inspired by HTTP in some ways, successfully disentangles transport and application layer security. Security and application state are independent of the underlying transport protocols. SIP also – in theory – supports end-to-end protection of messages but its partly S/MIME-based protection mechanisms have only seen limited implementation effort. But to properly decouple delivery infrastructure from content sources and thus reduce dependencies of the latter on the former, some similar mechanisms are needed.