Engineer

Protocol

the agreed rules two programs follow to talk

What it is

A protocol is a written agreement about a conversation between programs: what each message looks like, what it means, which side speaks first, what may follow what, and what either side does when the other says something unexpected. Two programs written by different people in different languages can work together because each was built against the same document rather than against the other.

NIST's glossary collects several definitions from its publications, and they share a core: rules covering the format of messages and the procedures around them, followed by two or more parties to reach a joint objective. One of them names the half that is easy to forget: the rules describe the order of the messages as well as their structure. A message in the right format at the wrong moment breaks the conversation as surely as a malformed one.

Specifications published as RFCs state their rules with capitalised keywords. MUST is an absolute requirement. SHOULD allows a deviation only when its full implications are understood and weighed. MAY marks something truly optional. Since 2017 those words carry that meaning only in capitals, so a lower-case "must" in the same document is ordinary English. The gap between SHOULD and MUST is the gap between a program that is unusual and one that is wrong.

Protocols are stacked. An application protocol, the part that knows about pages, menus or mail, usually rides on a transport protocol whose only job is to move bytes between two machines. TCP, the transport most of them use, provides what its specification calls "a reliable, in-order, byte-stream service": bytes arrive complete and in the order they were sent. It also promises no relationship between the chunks one side writes and the chunks the other side reads. A stream has no messages in it, so every protocol on top has to mark where its own

messages end, with a line ending, a length sent up front, the connection closing, or some other agreed marker. UDP makes the opposite bargain. It carries separate datagrams with a minimum of mechanism, and neither delivery nor protection against duplicates is guaranteed. Each transport tells the services on one machine apart by a port number, and the two number theirs separately.

The rest of a protocol's character comes from a few choices every design makes, on purpose or not:

- **Who speaks first.** Some servers greet the client, others wait in silence for a request, and when both sides wait, nothing happens until a timer runs out.
- **What the server remembers.** In a stateless protocol every request carries everything needed to answer it. In a stateful one, the meaning of a message depends on what came before it on the same connection.
- **How it grows.** Version numbers, option codes and fields that must be ignored when unknown are the room a protocol leaves for its own future.

Use the concept in a decision

Put "What is a protocol?" to work in a real decision. We can help relate the concept to the system you are building or buying. Wi-Fi, firewalls and remote access set up so staff can work from anywhere and only the right people get in.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Managed 30 Ubuntu Linux VPS instances — 30 Linux servers managed, with recovery planned for

Managed 30 Ubuntu Linux VPS instances Get in touch →

Why it matters

A protocol is what makes either end replaceable. Any client written to the document can talk to any server written to it, so a reader can change software and a server can be rewritten in another language without the other side noticing. Programs that agree only because they were built together cannot change apart.

The protocol also settles much of the security posture before any code exists. Which side opens the connection decides what has to be let in: a server that waits for clients needs a port open to the world, while a program that only dials out needs none. What the messages carry decides what can be logged, counted or leaked. A request that names a path can be logged by path, and hiding a

reader's network address does not prevent the server from counting requests. A protocol that runs without encryption shows every request to every network it crosses.

Finally, protocols outlive the programs that speak them. Implementations are replaced every few years while the conversation goes on for decades, and every behaviour deployed programs rely on becomes part of it, whatever the document says. The Internet Architecture Board has written about both halves of that problem: mistakes that are tolerated until they become the standard, and room for growth that nobody uses until it no longer works.

What it looks like in practice

This site answers on four protocols, out of one folder of files on one server: HTTP, Gemini, Gopher and an onion service over Tor. The page describes Gopher as a menu protocol from 1991 with plain TCP, no encryption and no scripting, and gives a whole request typed by hand:

```
printf '/contact\r\n' | nc engineer.company 70
```

It calls that line the entire protocol. Underneath, one deploy writes one folder and four front doors read it, so none of them is a separately maintained copy that could drift.

Firewall rules are verified by position rather than presence, because a packet filter evaluates an ordered list and stops at the first match. One rule of the check concerns protocols: exactly one rule exists per port, and always with a protocol, because a rule without one silently widens the surface. The declared public surface is a short list of five ports, asserted as a whole, so a port that appears without being declared fails the check.

The fourth door is an onion mirror, and its protocol shaped both its firewall and its reporting. The anonymity daemon reaches the network by making outbound circuits, so nothing dials in, no port opens, and neither firewall layer has anything to synchronise. Gemini requests and Gopher connections are tallied per day with no address and no path. The onion has no counter and no flag to add one, because this site deliberately leaves that audience uncounted.

Where it goes wrong

A tolerated mistake becomes the standard. The robustness principle, "be conservative in what you send, and liberal in what you accept", was meant to help programs work together. Applied without limit, it lets a flawed implementation ship and spread. The next implementation has to accept the same flaw to work with it, and the flaw becomes a de facto standard that anything new must be bug-for-bug compatible with. RFC 9413 puts it plainly: the deployed implementations become a conformance test that can supersede the specification. Reject what the specification forbids, as early in a protocol's life as possible, and make the error visible to someone who can fix it.

Two programs disagree about where a message ends. On a byte stream the boundary is wherever the protocol says it is. A protocol that offers two ways to say it, or a parser that accepts a sloppy version, lets a proxy and the server behind it read the same bytes as different messages. What one treats as the tail of a request, the other treats as a new request the proxy never checked. Allow one framing rule, reject anything ambiguous, and make every hop parse it the same way.

A rule names a port but not a protocol. A port number means nothing until a transport is named, and depending on the tool, a firewall rule written with only a number can open it on both TCP and UDP. The service that needed one of them gets it, and the other is open to whatever listens there later. Write the protocol into every rule, and compare the whole open surface against a declared list.

The room for growth stops working. If no deployed sender ever uses a protocol's extension points, receivers get written that reject what they have never seen, and the first real extension breaks against them. The longer such a receiver stays deployed, the harder it is to correct. Exercise the extension points from the start, including with values that mean nothing, so that tolerance is tested all the time rather than assumed.

Further reading

- NIST CSRC glossary: protocol — the definitions used in NIST security publications, each with the document it comes from.
- RFC 9293: Transmission Control Protocol — the current TCP standard, including the byte stream with no message boundaries.

- RFC 768: User Datagram Protocol — the whole of UDP in three pages, from 1980.
- RFC 1436: The Internet Gopher Protocol — an informational description of a whole application protocol, with sample sessions.
- RFC 9110: HTTP Semantics — the meaning of requests, responses, methods and status codes, independent of the version on the wire.
- RFC 9413: Maintaining Robust Protocols — the Internet Architecture Board on why tolerating errors entrenches them.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a protocol?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/protocol/