SAMPLE

Building the Communication Systems of Tomorrow

Designing Reliable Exchanges Across Networks, Devices, and People

Aiman JoeHanz

Free sample: Introduction & Chapter 1

putrabooks.github.io

Buy the full book:

Carousell carousell.sg/p/building-the-communication-systems-of-tomorrow-ebook-pdf-1465869650

ElevenReader

Ko-fi

Payhip

Contents

Introduction: What Must Survive the Journey

Chapter 1: The Message Is the Unit of Design

Chapter 2: Follow the Path, Then Follow the Dependencies

Chapter 3: Signals, Links, and the Physical Budget

Chapter 4: Transport Choices and Their Consequences

Chapter 5: Sessions, Voice, and Real-Time Media

Chapter 6: Messages That Outlive Connections

Chapter 7: Identity, Protection, and Data Boundaries

Chapter 8: Operate the Service You Actually Promised

Chapter 9: Build and Verify an End-to-End System

Chapter 10: What Tomorrow Must Prove

Connect

About this Series

This free sample contains the Introduction and Chapter 1. The complete book contains the full contents above.

Introduction: What Must Survive the Journey

A message can cross an ocean in a fraction of the time required to resolve a misunderstanding in the next room. That contrast captures both the achievement and the unfinished work of communication engineering. Moving a representation is one problem. Making that representation arrive in a useful, trustworthy, and actionable form is a larger one.

This book examines the larger problem. Its subject is the system that connects people, applications, and devices across imperfect links and imperfect organizational boundaries. Fiber, radio, and satellite links belong in that system, but so do addressing, session establishment, transport behavior, application agreements, security policy, operating procedures, and the person responsible when an exchange fails.

The intended reader need not design a radio or write a transport implementation. A project manager selecting a collaboration platform, a developer integrating a device fleet, and an engineer planning a resilient service all need an accurate picture of the dependencies between participants. Each must ask what information should move, what can go wrong along the way, and which party can detect or repair the resulting failure.

The argument begins with the message rather than with the network. A voice call, a batch transfer, and an equipment command have different requirements even when they traverse the same infrastructure. Audio that arrives too late can be unusable. A software image missing a small piece can be incomplete. A command received twice can trigger an unwanted second action unless the application is designed for repetition. Improving the connection alone does not resolve those differences.

Book 39 in this series examines 5G and 6G networks. This volume takes a different viewpoint. Mobile access appears as one possible part of an end-to-end path, not as the organizing subject. Likewise, cybersecurity, satellite engineering, and telecommunications policy are relevant boundaries, not invitations to reproduce entire neighboring books. The central question here is how a complete exchange works across those boundaries.

Protocol standards supply a factual foundation, but they do not make every architecture decision for an operator. A standard can define what an acknowledgment means without deciding whether that acknowledgment is enough for a payment, a maintenance request, or a clinical workflow. An implementation can comply with a protocol while its application still misunderstands stale data or duplicates an operation. This book separates protocol behavior from the additional promises that applications make.

A second distinction separates established mechanisms from future possibilities. TCP, IPv6, TLS 1.3, RTP, SIP, QUIC, and the Bundle Protocol are specified in the cited documents. Their existence does not establish universal deployment, automatic interoperability, or a guaranteed performance result. Future-facing proposals must be evaluated against workloads, operating conditions, and evidence. A compelling demonstration should not become a general claim without those qualifications.

The practical method is to make requirements observable. Identify the boundary at which an exchange counts as accepted. Identify the later boundary, if any, at which it counts as completed. State the acceptable delay and the validity period of the information. Decide whether duplication, loss, and reordering are tolerable. Specify what evidence permits the participants to distinguish success, rejection, and an unresolved outcome.

That method does not eliminate complexity. It puts complexity where people can inspect it. When a system relies on a queue, ask what happens to expired work. When it relies on a backup route, ask whether the route shares power or management dependencies with the primary route. When it encrypts traffic, ask which participants terminate the encryption and what those participants are allowed to see. When it advertises low latency, ask which portion of the complete interaction was measured.

Throughout the book, hypothetical examples illustrate these questions without pretending to document real deployments. Numerical examples are calculations under stated assumptions, not vendor benchmarks. References identify the standards behind specific protocol claims. Operational advice is presented as engineering guidance to evaluate, not as a guarantee that one design will suit every organization.

The communication systems of tomorrow will not be judged only by the signals they can carry. They will be judged by whether people can rely on the exchanges that those signals support. The first step toward building them is to decide what must survive the journey.

Chapter 1: The Message Is the Unit of Design

Imagine a building-management company receiving temperature reports from several properties. Every site has an internet connection. Every controller can establish a session with the central service. The dashboard shows green connection indicators. Yet a technician discovers that one building has been displaying yesterday's readings all morning.

This is a hypothetical design example, not a report of a particular incident. It illustrates a distinction that will recur throughout this book: being connected does not establish that useful communication has occurred. The controller might have sent a report that never reached the application. The application might have stored it without updating the display. The dashboard might have received an old report again and mistaken its arrival time for its measurement time. None of those failures requires a broken cable.

A communication system carries something that matters from one participant to another. That something might be spoken audio, a written instruction, a sensor observation, a software update, or a request to release a payment. The system's purpose is not simply to keep a link active. It must preserve the properties that make the exchange useful: the right information, an intelligible representation, an appropriate recipient, an acceptable delay, and enough evidence to decide what happened.

Start a design discussion with those properties rather than with a shopping list of equipment. A radio, switch, messaging broker, and conferencing platform can all be appropriate purchases. None is a substitute for deciding what counts as success. When the success condition remains vague, each supplier can meet its own specification while the overall service disappoints its users.

For the building example, a useful success condition might be that the central application displays each site's latest valid observation, makes the observation's age visible, and alerts an operator when the information is too old for the intended decision. That condition differs from a promise that a certain percentage of packets will arrive. It also differs from a promise that the dashboard will always be reachable. The application can be reachable while showing stale information.

The starting question is therefore concrete: what must a recipient be able to know or do after the exchange?

Engineers use the word delivered at several boundaries. Those meanings are related but not interchangeable.

At a physical boundary, delivery can mean that a receiver detected a signal. At a network boundary, it can mean that data reached a destination address. At a transport boundary, it can mean that a protocol acknowledged receipt of bytes. At an application boundary, it can mean that the receiving program accepted an operation. A human-facing service adds another boundary: someone may need to read, understand, or act on the result.

TCP specifies a reliable, ordered byte-stream service. Its acknowledgments support the transport's management of that stream; they are not receipts for business decisions. An application must define what acceptance means above TCP. RFC 9293 is the reference for the TCP behavior discussed here.

Consider a maintenance request sent to a work-order service. The sender transmits the request and then loses its connection before receiving a response. Several outcomes are possible. The server may never have received the complete request. It may have received the request but rejected it. It may have accepted and stored the work order, with only the response being lost. A connection failure by itself cannot distinguish those outcomes.

If the sender retries without an operation identifier, it might create a second work order. If it refuses to retry, it might leave the job unrecorded. The network has exposed an uncertainty that application design must resolve. An operation identifier, a durable record of accepted operations, and a way to query the operation's state can make a retry safe within a stated retention policy. The policy matters: if the receiving service forgets old identifiers, a sufficiently late retry can become a new operation.

This is not a demand that every application build a sophisticated transaction engine. A casual status update may tolerate duplication. A command to dispense a physical item may not. The required mechanism follows from the consequence of ambiguity.

A diagram of the system should label its acknowledgment boundaries explicitly. Replace an unlabeled arrow saying success with a statement such as bytes acknowledged, request validated, record committed, recipient notified, or operator acknowledged. These labels expose gaps that are easy to miss when a single check mark stands for the whole exchange.

A message needs a meaning that survives its journey. Encoding gives that meaning a form that a machine or person can transmit and interpret. The same numerical value can be encoded as text, a binary integer, or a field within a structured document. Encoding choices affect size, interoperability, precision, and the effort required to inspect failures.

Before choosing an encoding, identify the interpretation that must remain

stable. A temperature observation needs a unit. A timestamp needs a stated relationship to time. A device identifier needs a scope. A command needs a description of the requested state or action. Omitting those details can produce a message that is syntactically valid but operationally wrong.

Suppose a hypothetical controller reports a temperature value of 24. A receiver that assumes degrees Celsius reaches a different conclusion from one that assumes degrees Fahrenheit. Neither interpretation can be recovered reliably from the number alone. A documented schema can establish a fixed unit, or the unit can travel with the value. The choice depends on the participants and the likelihood of change, but the agreement must exist somewhere that implementers can find and test.

Identifiers deserve similar attention. A device called controller-7 may be unique within one installation and repeated across several others. Combining an installation identifier with a local identifier may be sufficient. A globally unique identifier can be convenient, but it does not automatically answer who owns the device or whether it is still authorized to send reports.

Time also carries several meanings. The event time describes when an observation or action occurred. The send time describes when a participant transmitted its message. The receive time describes when another participant received it. The processing time describes when an application handled it. A stored record may need more than one of these values.

Do not overwrite event time with receive time merely because receive time is easier to measure. A delayed observation may arrive now while describing a condition that existed an hour earlier. Conversely, a clock error may make an otherwise fresh observation appear to come from the future. The application needs a policy for those possibilities rather than a hidden assumption that all clocks agree.

A message contract states what participants may send, what a receiver must check, and what happens when the agreement cannot be met. It is larger than a field list but smaller than a complete implementation.

For the building-management example, a candidate observation contract could identify the site and sensor, name the observed quantity, define its unit and numerical representation, include an observation time, assign an observation identifier, and specify how quality information is represented. It could state whether a later message replaces an earlier one and how the service marks an observation stale.

The contract should not claim that a timestamp proves the clock was correct. It should distinguish a timestamp reported by a device from an independently

established time reference. Similarly, a sensor value should not imply a degree of precision that the measuring arrangement cannot support. Representation precision and measurement accuracy are different properties.

Validation has several layers. Structural validation checks whether a message contains the expected fields and types. Semantic validation checks relationships among those fields: a quantity and unit must agree, for example.

Authorization checks whether the sender is permitted to submit the information or request the action. Operational validation checks whether the message remains usable in the current context.

A delayed command illustrates the last layer. The command may be well formed, correctly signed, and sent by an authorized participant, yet no longer be appropriate. A request to open a gate during a loading appointment should not necessarily open it after the appointment has ended. An expiry rule is part of the message's meaning, not an optional optimization.

State the rejection behavior too. A receiver might reject a malformed message immediately, quarantine it for investigation, or accept it with a visible quality warning. Quietly substituting default values can be dangerous when those defaults have physical or financial consequences. If the service substitutes a value, the substitution should be an intentional and observable part of the contract.

A sequence number tells a receiver something about order within an agreed sequence. It does not, by itself, tell the receiver how old a message is. A timestamp can help describe age, but it does not, by itself, establish a trustworthy ordering across participants whose clocks differ.

Within a single sensor stream, a sequence number can help reveal missing observations. Across several sensors, independent sequences do not establish a common order. Across a restart, the same sensor might begin numbering again. A design can distinguish sessions or persist a counter, but either choice needs to account for failure and replacement.

Return to the dashboard that displays old readings. If it treats the last arriving message as the latest observation, a delayed message can overwrite a newer one. Comparing observation times can improve this behavior when the clocks are sufficiently trustworthy. Comparing a per-sensor generation and sequence can improve it when devices maintain those values correctly. Some systems need both, with an explicit policy for disagreements.

Ordering requirements should be scoped narrowly. A control channel may need ordered commands for one actuator without needing a single order for every

event in the company. Creating a universal order where the application does not require one can introduce a bottleneck and a new dependency. Refusing all ordering mechanisms can be equally unhelpful when action sequences affect safety.

A useful design statement names the entity to which order applies and describes what the receiver does when that order cannot be established. For example, a receiving service might accept observations out of order but refuse to execute a command whose predecessor is unresolved. These are different policies for different message classes, not contradictions.

A communication system often carries several traffic classes over shared infrastructure. Classifying them by consequence helps explain why a single transport policy may not serve them all.

An interactive conversation is sensitive to delay. An audio segment that arrives after its intended playback time may be less useful than a short gap. A file transfer usually values completeness more than immediate arrival of every individual segment. A safety-related command may value authenticated intent, a bounded lifetime, and clear acknowledgment more than aggregate throughput. A background observation stream may tolerate brief gaps but still require freshness indicators.

Those differences do not prescribe one universal protocol mapping. They guide the questions an implementation must answer. RTP, for instance, provides functions for transporting real-time media, and RTCP provides associated control and monitoring. RFC 3550 explicitly does not promise resource reservation or quality of service merely because RTP is in use. An application and its operating environment must still manage playback, congestion, and failure.

The same caution applies to queueing systems. A queue can decouple a sender from a receiver's temporary unavailability. It can also preserve work long past the time when the work remains useful. A command with a short validity period should not wait indefinitely behind a large backlog of low-priority reports. A system that needs both durable history and timely action may need separate policies, even when it uses the same software platform for both.

Define what to discard as carefully as what to retain. For repeated observations of a current state, retaining only a recent observation might be appropriate for one consumer. An audit consumer may need the full sequence. Do not make one storage choice stand silently for both requirements.

A service deadline must account for more than propagation through a cable or

radio link. Encoding, queueing, transport recovery, receiving-side processing, storage, and presentation all consume time. A component that meets its individual target can still contribute to an end-to-end failure if the targets do not fit together.

Take an explicitly hypothetical budget: a user needs confirmation of a routine command within two seconds. The sending application spends 100 milliseconds validating and preparing the request. Network transit and queueing consume 200 milliseconds. The receiving service spends 300 milliseconds on authorization and state checks, then 500 milliseconds obtaining a dependency's response. Returning and displaying the result takes another 200 milliseconds. Those assumed components total 1,300 milliseconds, leaving 700 milliseconds for variation within the example's deadline.

The calculation is only an illustration. It is not a benchmark for any product or network. Its purpose is to make a hidden remainder visible. If the dependency sometimes takes an additional second, the system misses the deadline even though its network connection is fast. If retry policies add a delay larger than the remaining budget, a retry cannot rescue that particular attempt in time.

Specify whether a deadline includes acknowledgment or completion. A receiver can acknowledge acceptance quickly while finishing work later. That may be appropriate for a report export. It may not satisfy a user waiting for confirmation that a physical action has occurred. Separating accepted, in progress, completed, and failed can let an application give a truthful response without pretending that acceptance means completion.

Distribution matters as well as an average. A service whose mean response time is acceptable can still fail a consequential minority of interactions. Select a measurement that corresponds to the promised experience and state the observation window and workload. Without those qualifications, a latency figure can conceal the conditions in which it was obtained.

Troubleshooting a distributed conversation requires evidence that can be related across boundaries. An operation identifier can connect a sender's attempt with the receiving service's decision. A message identifier can connect a stored observation with its ingestion record. A transport connection identifier can help investigate a connection, but it may not be sufficient for a business operation that spans several connections.

Choose identifiers for the questions investigators must answer. Did the receiver see this request? Did it reject it? Was the operation committed? Was a response created? Did the client observe that response? Which version of a

message schema was used? Evidence for one question should not be treated as evidence for all the others.

Record only the data needed for the investigation and permitted by the system's data-handling rules. A debugging log does not need to become an uncontrolled copy of every conversation. Message content, identifiers, and timing can all be sensitive in context. Redaction, access controls, and retention limits belong in the design rather than in a cleanup exercise after a failure.

Observability also changes system behavior. Logging every event synchronously can impose delay or create a storage bottleneck. Dropping all logs when a service becomes busy can remove the evidence precisely when investigators need it. Select an operating policy that states what may be sampled, buffered, or lost, and identify the evidence that must survive.

A success metric should align with the acknowledgment boundary it measures. Count accepted commands separately from completed commands. Count observations ingested separately from observations displayed within the freshness limit. If these measurements disagree, the disagreement is useful evidence, not a reporting inconvenience to be hidden.

A participant can fail after receiving a message and before preserving it. A service can preserve an operation and fail before responding. A connection can disappear without either participant immediately knowing why. An intermediary can retain data longer than the sender expects. Every one of these possibilities affects the message contract.

The design task is not to promise that no boundary will ever fail. It is to establish the system's behavior when a boundary does fail. Can a participant retry? Can it discover the outcome? Is a retry safe? Is the old message still valid? Does an operator need to intervene? Is it acceptable to stop the workflow until the ambiguity is resolved?

For a noncritical observation, losing one report may be acceptable if later reports restore a useful current view. For an operation that changes a scarce physical resource, unresolved state may require a reconciliation process. The same network outage can therefore have different consequences for different applications.

Make the reconciliation path a first-class part of the design. A secondary status query, a durable acceptance receipt, or an operator workflow may provide a better recovery mechanism than endless automated retries. Automated recovery is valuable when the system can determine what it is doing. When it cannot, repeated action can amplify a small uncertainty into a larger incident.

A practical first artifact for a new communication service is a short contract table. Each row names a message class. The columns identify its sender and receiver, interpretation, authorization rule, freshness requirement, ordering scope, acceptance boundary, retry rule, and evidence needed for investigation.

For a hypothetical equipment-monitoring service, observations and commands would occupy different rows. Observations might be accepted out of order and retained as history, while the current display uses a freshness rule. Commands might require an operation identifier and expiry time, with the receiver recording whether it accepted or rejected each attempt. Notifications to operators would have their own acknowledgment boundary because displaying a notification does not establish that an operator has read it.

Review the table with people who own the consequences, not only with people who administer the network. A facilities manager can explain when a stale observation becomes unusable. A maintenance technician can explain whether duplicated work orders create confusion or expense. A security reviewer can identify actions that require stronger authority checks. An application developer can describe which outcomes the software can actually distinguish.

Then test the contract against deliberate ambiguity. Remove a response. Delay an observation. Repeat a command. Restart the sender. Change a schema version. Make a dependency unavailable. These tests do not require a perfect model of every future failure. They expose whether the stated behavior exists at all.

The resulting table does not replace a network architecture. It gives that architecture a purpose. Once the system knows what its messages mean, which delays matter, and what evidence must survive, choices about links, transports, intermediaries, and redundancy become easier to evaluate. Without that foundation, tomorrow's communication system risks becoming a faster way to deliver uncertainty.

End of free sample. The full book is available at putrabooks.github.io

ABOUT THE AUTHOR

Aiman JoeHanz writes the Building of Tomorrow series -- structural, vendor-neutral guides to the technologies reshaping industry, labor, and daily life. Each volume treats one technology the way an engineer treats a machine: definition first, then anatomy, failure modes, principles, build, deployment, and the honest decade ahead.