Idempotence Is Not a Medical Condition

An essential property for reliable systems

Pat Helland

The definition of *distributed computing* can be confusing. Sometimes, it refers to a tightly coupled cluster of computers working together to look like one larger computer. More often, however, it refers to a bunch of loosely related applications chattering together without a lot of system-level support.

This lack of support in distributed computing environments makes it difficult to write applications that work together. Messages sent between systems do not have crisp guarantees for delivery. They can get lost, and so, after a timeout, they are retried. The application on the other side of the communication may see multiple messages arrive where one was intended. These messages may be reordered and interleaved with different messages. Ensuring that the application behaves as intended can be very hard to design and implement. It is even harder to test.

In a world full of retried messages, *idempotence* is an essential property for reliable systems. *Idempotence* is a mathematical term meaning that performing an operation multiple times will have the same effect as performing it exactly one time. The challenges occur when messages are related to each other and may have ordering constraints. How are messages associated? What can go wrong? How can an application developer build a correctly functioning app without losing his or her mojo?

THE MESSAGING ENVIRONMENT

This section frames the problem by describing the kind of application under consideration. It looks at services and how long-running work can run across them, and it considers the need for messaging across applications that evolve independently in a world where they share relatively simple standards.

SERVERS AND SERVICES

This article considers messaging in a service-oriented environment consisting of a collection of servers that share the work. The hope is that you can scale by adding more servers when the demand increases. This opens up the question of how later messages can locate a service (running on a server) that remembers what happened before.

MESSAGES AND LONG-RUNNING WORK

Sometimes related messages arrive much later, perhaps days or weeks later. These messages are part of the same piece of work performed by an application attempting to work across multiple machines, departments, or enterprises. Somehow the communicating applications need to have the information necessary for correlating the related messages.

Predictable and consistent behavior is essential for message processing, even when some of the participants have crashed and restarted. Either the earlier messages didn't matter or the participant needs to remember them. This implies some form of durability capturing the essence of what was important about the earlier messages so the long-running work can continue. Some systems do need the information from the earlier message to process the later ones but don't make provisions for remembering the stuff across system crashes.

Of course, there is a technical term for unusual behavior when a system crash intervenes; it is called a *bug*.

BUILDING APPLICATIONS IN AN EVER-CHANGING WORLD

The shape and form of applications continue to evolve as years go by, transitioning from mainframes to minicomputers to PCs to departmental networks. Now scalable cloud-computing networks offer new ways of implementing applications in support of an ever-increasing number of messaging partners.

As the mechanisms for implementing applications change, the subtleties of messaging change. Most applications work to include messaging with multiple partners using semi-stable Internet standards. These standards support the evolving environment but cannot eliminate some of the possible anomalies.

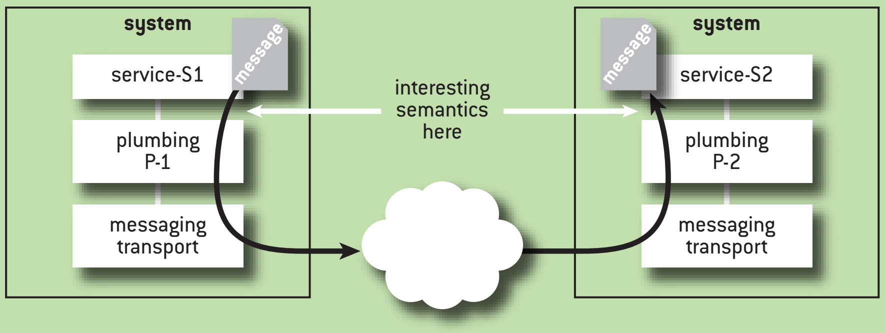
IMAGINING A DIALOG BETWEEN TWO SERVICES

This article considers the challenges of communicating between two parties. (Other messaging patterns, such as pub-sub, are not addressed here.) It imagines an arbitrary communication sequence of messages between these two parties that are related to each other and assumes that somehow the two programs have a notion of the work accomplished by this dialog. Even with the best "messaging plumbing" support, a two-party messaging dialog between two services has complications to be addressed.

PLUMBING AND APPLICATIONS

Applications often run on top of some form of "plumbing." Among other services, the application's plumbing offers assistance in the delivery of messages. It likely offers help in naming, delivery, and retries, and may provide an abstraction that relates messages together (e.g., request-response).

FIGURE 1 **Applications Talk to Their Local Plumbing**



As an application designer, you can only understand what *you* see as you interact with the plumbing on your client, server, or mobile device. You can anticipate or infer what happens farther down the line, but all of it is intermediated by your local plumbing (see figure 1).

IDENTITY AND RELATIONSHIPS

When a local application (or service) decides to engage in a dialog with another service, it must use a name for the intended partner. The identity of a partner service for the first message specifies the desire to chat with a service doing the application's work but not yet engaged in some joint project. That's really different from connecting to the same service instance that has processed previous messages.

A relationship is established when messages are sent in a dialog. Processing subsequent messages implies that the partner remembers earlier messages. This relationship implies an identity for the partner in the middle of the dialog that is different from a fresh partner at the beginning of a dialog.

The representation of the mid-dialog identity, and how the later messages are routed to the right place, are all part of the "plumbing." Sometimes the plumbing is so weak that the application developer must provide these mechanisms, or the plumbing may not be universally available on all the communicating applications, so the applications must delve into solving these issues. For now, let's assume that the plumbing is reasonably smart, and look at the best-case behavior an application can expect.

MESSAGES, DATA, AND TRANSACTIONS

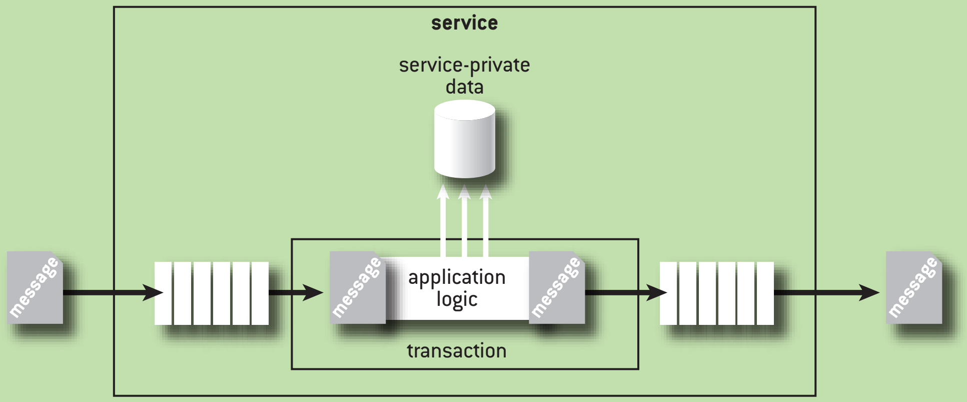
What happens when the application has some data it is manipulating (perhaps in a database with transactional updates)? Consider the following options:

• Message consumption is recorded *before* processing. This is rare because building a predictable application is very difficult. Sometimes the message is recorded as consumed, the application sets out to do the work, and the transaction fails. When this happens, the message is effectively not delivered at all.

• The message is consumed as part of the database transaction. This is the easiest option for the

FIGURE 2

Plumbing Ties the Incoming Message to Database Changes and Outgoing Messages



application, but it is not commonly available. All too often, the messaging and database systems are separate. Some systems, such as SQL Service Broker[4] provide this option (see figure 2). Sometimes, the plumbing will transactionally tie the consumption of the incoming message to changes in the application's database and to outgoing messages. This makes it easier for the application but requires tight coordination between messaging and database.

• The message is consumed *after* processing. This is the most common case. The application must be designed so that each and every message is idempotent in its processing. There is a failure window in which the work is successfully applied to the database, but a failure prevents the messaging system from knowing the message was consumed. The messaging system will re-drive the delivery of the message.

DON'T TALK AND LISTEN AT THE SAME TIME

Outgoing messages are typically queued as part of committing the processing work. They, too, may be either tightly coupled to the database changes or allowed to be separate. When the database and messaging changes are tied together with a common transaction, the consumption of the message, the changes to the database, and the enqueuing of outgoing messages are all tied together in one transaction (see figure 2). Only *after* enqueuing an outgoing message will the message depart the sending system. Allowing it to depart before the transaction commits may open up the possibility of the message being sent but the transaction aborting.

IN-ORDER DELIVERY: HISTORY OR CURRENCY?

In a communication between two partners, there is a clear notion of sending order in each direction. You may be sending to me while I am sending to you, meaning there is fuzziness in the ordering for messages going past each other, but only one message at a time is sent in a specific direction.

A listener can easily specify that it doesn't want out-of-order messages. If you sent me *n* different messages in a specific order, do I really want to see message *n-1* when I've already seen message *n*? If the plumbing allows the application to see this reordering, then the application very likely has to add some extra protocol and processing code to cope with the craziness.

Suppose the messages *always* come in order. There are two reasonable application behaviors:

HISTORY (NO GAPS). Not only will the messages be delivered in order, but also the plumbing will respect the sequence and not allow gaps. This means that the plumbing will not deliver message *n* in the dialog sequence to the app if message *n-1* is missing. It may work to get message *n-1* under the covers, but that is transparent to the application. This is very useful for business process workflows and many other cases. Losing a message would make the application's design very, very difficult, so the app wants no gaps.

CURRENCY (DELIVER THE LATEST—GAPS OR NO GAPS). Sometimes the delay to fill in the gap is more of a problem than the gap. When watching the price for a specific stock or the temperature gauge of a chemical process in a factory, you may be happy to skip a few intermediate results to get a more timely reading. Still, order is desired to avoid moving back in time for the stock price or temperature.

KNOWING WHAT YOU DON'T KNOW WHEN SENDING MESSAGES

When a communicating application wants to request some work to be done by its partner, there are a few stages to consider:

BEFORE YOU SEND THE REQUEST. At this point, you are very confident the work hasn't been done.
AFTER YOU SEND BUT BEFORE YOU RECEIVE AN ANSWER. This is the point of confusion. You have absolutely no idea if the other guy has done anything. The work may be done soon, may already be done, or may never get done. Sending the request increases your confusion.
AFTER YOU RECEIVE THE ANSWER. Now, you know. Either it worked or it didn't work, but you are less confused.

Messaging across loosely coupled partners is inherently an exercise in confusion and uncertainty. It is important for the application programmer to understand the ambiguities involved in messaging (see figure 3). The interesting semantic occurs as an application talks to the local plumbing on its box. This is all it can see.

I'M NOT WAITING FOREVER!

Every application is allowed to get bored and abandon its participation in the work. It is useful to have the messaging plumbing track the time since the last message was received. Frequently, the application will want to specify that it is willing to wait only so long until it gives up. If the plumbing helps with this, that's great. If not, the application will need to track on its own any timeouts it needs.

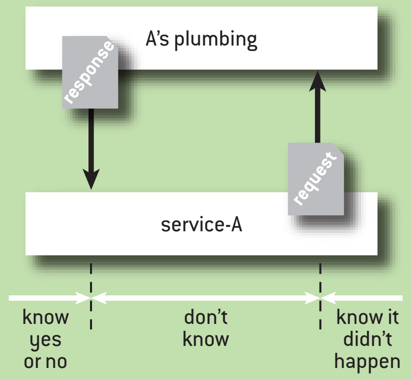
Some application developers may push for no timeout and argue it is OK to wait indefinitely. I typically propose they set the timeout to 30 years. That, in turn, generates a response that I need to be reasonable and not silly. *Why is 30 years silly but infinity is reasonable?* I have yet to see a messaging application that really wants to wait for an unbounded period of time…

WHEN YOUR PLUMBER DOESN'T UNDERSTAND YOU

Nothing beats a plumber who can alleviate your worries and make everything "just work." It is especially nice if that plumber shares the same understanding of you and your needs.

Many applications just want to have a multimessage dialog between two services in which each

FIGURE 3 **The Interesting Semantic Occurs as an Application Talks to the Local Plumbing on Its Box**



accomplishes part of the work. I've just described what can be expected in the best case when you and your plumber share clear notions of:

• How you should talk to your plumbing.
• Who you are talking to on the other side.
• How transactions work with your data (and incoming and outgoing messages).
• Whether messages are delivered only after all previous messages or you skip messages to get the latest.
• When the sender knows a message has been delivered and when it is ambiguous.
• When a service may abandon the communication and how the partner learns about it.
• How you test the timeout.

These challenges assume that you have met the plumber of your dreams who has implemented great support for your messaging environment. It is rarely that clean and simple. On the contrary, the application developer needs to watch out for a slew of issues.

GUARANTEED MESSAGE DELIVERY

Some messaging systems offer guaranteed delivery. These systems (e.g., MQ-Series)[1] will typically record the message in a disk-based queue as a part of accepting the send of the message. The consumption of the message (and its removal from the queue) happens either as part of the transaction stimulated by the message or only after the transactional work has been completed. In the latter case, the work may be processed twice if there is a glitch.

One challenge in the classic guaranteed-delivery queue system occurs when the application gets a message that it can't process. Guaranteed delivery means the messaging system delivered it, but there's no guarantee that the message was well formed or, even if it was, that the tempestuous application did something reasonable with the message. Before my wife started doing our household bills and it was my responsibility, the reliable delivery of the electric bill to our house was only loosely correlated to the electric company receiving its money.

ZERO OR MORE TIMES... GUARANTEED!

When considering the behavior of the underlying message transport, it is best to remember what is promised. *Each message is guaranteed to be delivered zero or more times!* That is a guarantee you can count on. There is a lovely probability spike showing that most messages are delivered one time.

If you don't assume that the underlying transport may drop or repeat messages, then you will have latent bugs in your application. More interesting is the question of how much help the plumbing layered on top of the transport can give you. If the communicating applications run on top of plumbing that shares common abstractions for messaging, some help may exist. In most environments, the app must cope with this issue by itself.

WHY ISN'T TCP ENOUGH?

TCP has had a major impact on unifying the ways in which we perform data communication.[5] It offers exactly-once and in-order byte delivery *between two communicating processes*. It offers no guarantees once the connection is terminated or one of the processes completes or fails. This means that it covers only a small portion of the landscape visible to developers building reliable applications in a loosely coupled distributed system. Realistically, the application layers on top of TCP and must solve many of the same problems all over again.

Requests get lost, so just about every messaging system retries transmitting. The messaging system often uses TCP, which has its own mechanism to ensure the reliable delivery of bytes from process to process. TCP's guarantees are real but apply only to a single process chatting with exactly one other process. Challenges arise when longer-lived participants are involved.

Consider, for example, HTTP Web requests. HTTP typically shuts down the TCP connection between requests. When a persistent HTTP connection is used, the TCP connection is typically left alive, but there is no guarantee. This means that any use of HTTP on top of TCP may result in multiple sends of the HTTP request. For this reason, most HTTP requests are idempotent.[3]

In scalable Web-service worlds, we are constantly reimplementing the same sliding window protocol[2] that is so ubiquitous in TCP. In TCP, the endpoints are running processes. The failure of either of the processes means the failure of the TCP connection. In a long-running messaging environment implemented by a collection of servers, the semantics of the endpoint are more complex. As the representation of an endpoint and its state evolves, so do the messaging anomalies that are (hopefully) managed by the plumbing. More likely, they are incrementally solved by the application as patches to surprising bugs. Either way, even application developers will need to have a copy of Andrew Tanenbaum's classic book, *Computer Networks,* at their fingertips.[2]

AVOIDING EMBARRASSMENT WHEN TALKING ABOUT IDEMPOTENCE

To review, idempotence means that multiple invocations of some work are identical to exactly one invocation.

- Sweeping the floor is idempotent. If you sweep it multiple times, you still get a clean floor.
- Withdrawing $1 billion is not idempotent. Stuttering and retrying might be annoying.
- Processing withdrawal XYZ for $1 billion if not already processed is idempotent.
- Baking a cake is not idempotent.
- Baking a cake starting from a shopping list (if you don't care about money) is idempotent.
- Reading record X is idempotent. Even if the value changes, any legitimate value for X during the window between the issuance of the read and the return of the answer is correct.

The definition of idempotent in computer usage is: "Acting as if used only once, even if used multiple times." While this is true, there are frequently side effects of the multiple attempts. Let's consider a few side effects that are not typically considered semantically relevant:

HEAPS. Imagine a system that uses a heap during the processing of a request. You would naturally expect that the heap might become more fragmented with multiple requests.

LOGGING AND MONITORING. Most server systems maintain logs that allow analysis and monitoring of the system. Repeated requests will influence the contents of the logs and the monitoring statistics.

PERFORMANCE. The repeated requests may consume computation, network, and/or storage resources. This may be a tax on the throughput of the system.

These side effects are not relevant to the semantics of the application behavior, so the processing of an idempotent request is still considered idempotent even if side effects exist.

CHALLENGES OF MULTIMESSAGE INTERACTIONS

Any message may arrive multiple times… even after a *long* while. Think of a messaging system as containing a bunch of Machiavellian gnomes who are watching your messages float by so they can interject a copy of a message at precisely the worst time for your application (see figure 4). In most loosely coupled systems, messages may arrive multiple times. Furthermore, related messages may be delivered out of order.

STATEFULNESS AND MULTIMESSAGE INTERACTIONS

A typical approach to this problem is to use request-response and then ensure the messages processed are idempotent. This has the benefit of the application seeing the delivery of the message via a positive response. If the application gets a response from its intended partner, then it is confident that the message has actually arrived. Because of retries, the receiving application may receive the message many times.

This challenge is further compounded when multiple messages are involved in making a piece of longer-running work happen. The duration of the work and the allowable failures for the work must be considered, as in the following cases:

LIGHTWEIGHT PROCESS STATE. Sometimes the failure of a process at either end of the shared computation can cause the entire workflow to be abandoned. If that's the case, then the intermediate state can be kept in the running process.

LONG-RUNNING DURABLE STATE. In this case, the long-running work must continue even if one of the partner systems crashes and restarts. This is common for business processes that may last days or weeks but still have a meaningful messaging interaction.

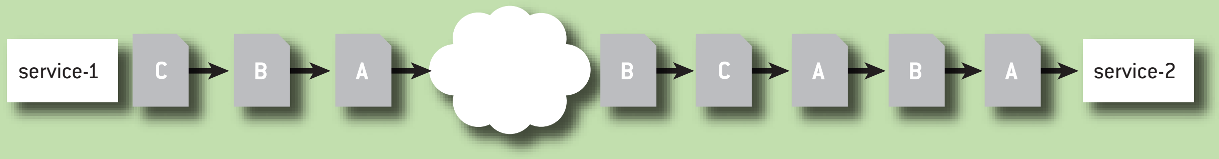
STATELESS APPLICATION SERVERS. Many architectures have the computation separated from the state. The state may be kept in a durable database on a single server, replicated across a bunch of durable data stores, or stored in any number of other novel ways.

In all of these cases, the system must behave correctly in bringing together the state (which is the result of memories from earlier messages) with the new message. Messages may be merged into the state in different orders and those orders may be affected by failures of one or more of the systems.

YOU'RE JUST NOT THE PERSON I THOUGHT YOU WERE

Many systems implement a target application with a pool of load-balanced servers. This works well when the incoming message makes no assumptions about previous communications and previous

FIGURE 4 **In Most Loosely-Coupled Systems, Messages May Arrive Multiple Times**



state. The first message is routed to one of the servers and gets processed (see figure 5). When communicating with Service Foo, multiple machines may be implementing the service. This puts interesting burdens on the implementation of Service Foo and can show up in anomalous behavior visible to the communicating partner.

Challenges may arise when the second (or later) message arrives and expects the partner not to have amnesia. When chatting with multiple messages, you assume your partner remembers the earlier messages. The whole notion of a multimessage dialog is based on this.

WHAT DO YOU MEAN THE WORK IS NOT DONE ON THE PREMISES?

When Service A talks to Service B, you don't know where the work is really done. Service A believes it is doing work with Service B, while Service B may actually subcontract all the work to Service C (see figure 6). This is not in itself a problem, but it can magnify the failure possibilities seen by Service A.

You cannot assume that the work is actually done by the system you are chatting with. All you know is that you have a messaging protocol, and, if things are going well, appropriate messages come back from the named partner according to the protocol's definition. You simply don't know what goes on behind the curtain.

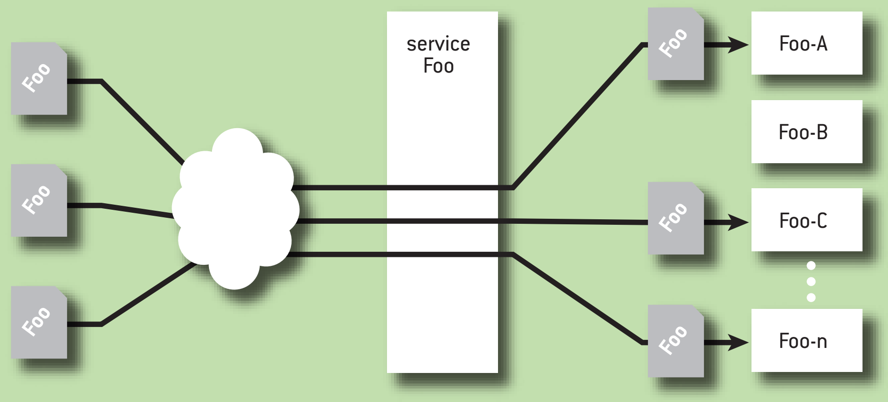
ACK MEANS "STOP REPEATING YOURSELF"

You know a message has been delivered only when the answer comes back from the partner doing the work. Tracking the intermediate waypoints doesn't help to know that the work will get done. Knowing a FedEx package has reached Memphis won't tell you that your grandmother will receive her box of chocolates.

When a messaging transport sends an ACK to a sender, it means that the message has been received at the next machine. It says nothing about the actual delivery of the message to the destination and even less about any processing the application may do with the message. This is even more complicated if there is an intermediate application that subcontracts the work to another

FIGURE 5

Multiple Machines May Implement the Service via Service-Foo



application service (see figure 7). Transport and plumbing acknowledgments cannot be visible to applications or bugs may be introduced when the destination service is reconfigured. The ACK tells Service A's plumbing that Service B's plumbing has the message, but does not tell Service A anything about Service C's receipt of the message. Service A must not act on the ACK.

ACK means sending the message again won't help. If the sending application is made aware of the ACK and acts on that knowledge, then it may cause bugs if and when the real work fails to materialize.

PLUMBING CAN MASK AMBIGUITIES OF PARTNERSHIP (BUT RARELY DOES)

It is possible for the message-delivery plumbing to have a formalized notion of a long-running dialog. The plumbing must define and implement the following:

STATE. How is the state information from a partially completed dialog represented by the application? Remember, either the state must survive failures of a component or the dialog must

FIGURE 6 **Service Behavior Is in the Eye of the Beholder**

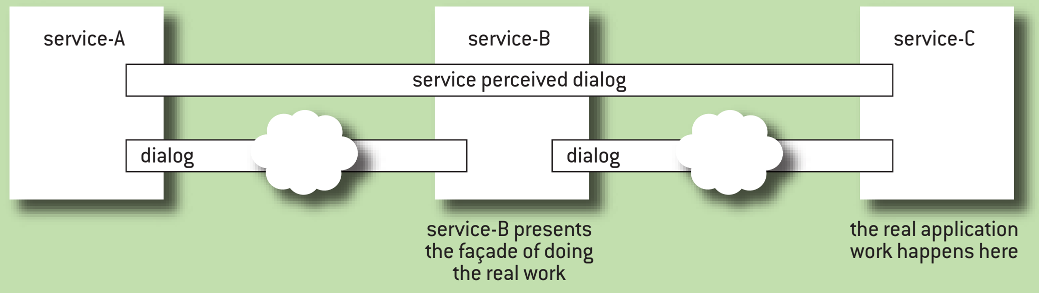
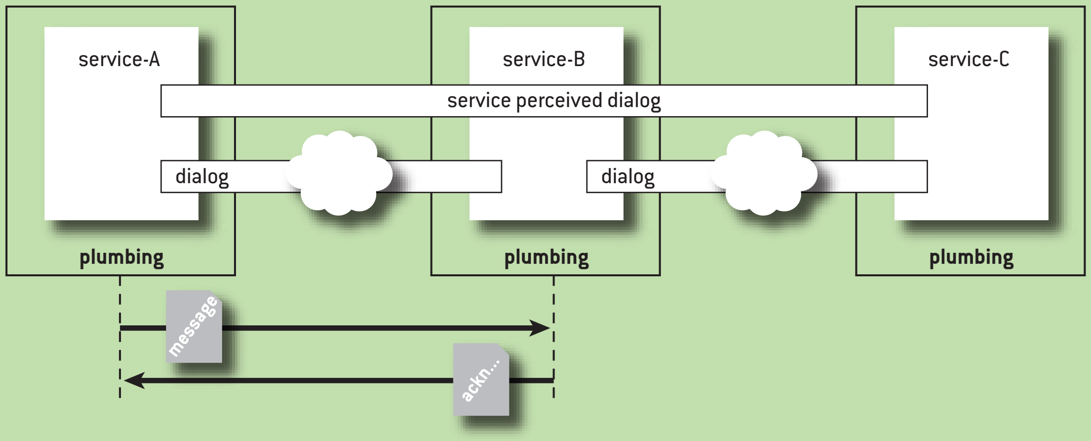

service-A
service-B
service-C
service perceived dialog
dialog
dialog
service-B presents the façade of doing the real work
the real application work happens here

FIGURE 7 **Transport and Plumbing Acknowledgements Should Not Be Visible to Applications**



cleanly fail as a consequence of the failure—just forgetting the early messages is bad.
ROUTING. How do later messages find a service (and server) that can locate the state and correctly process the new message without amnesia about the earlier messages?
DIALOG SEMANTICS. How can the dialog provide correct semantics even when the real work of the dialog is subcontracted to somebody way the heck over yonder? Part of this is properly masking transport issues (such as ACKs) that will only cause problems if seen by the application.

Thus, it seems that messaging semantics are intimately tied to naming, routing, and state management. This inevitably means that the application designer is faced with challenges when the database comes from a plumbing supply house that is different from that of the messaging system. One system that does provide crisp dialog semantics is SQL Service Broker.[4] It does so by holding the messaging and dialog state in the SQL database.

TALKING TO A SERVICE

Services are designed to be black boxes. You know the service address, start working with it, chatter back and forth, and then finish. It turns out that, even with the support of some plumbing to provide you with dialogs, there are issues with repeated messages and lost messages. These challenges are compounded when the target service is implemented in a scalable fashion. Of course, that means you may start out interacting with a service that is not yet scalable, and, as it grows, certain new obstacles may arise.

STAGES OF A DIALOG

A dialog goes through three stages in its lifetime:
INITIATION. Messages are sent from the dialog initiator to the target of the dialog. The initiator doesn't know if the target is implemented as a single server or a load-balanced pool.
ESTABLISHED. Messages may flow in both directions in full-duplex fashion. The messaging system has now ensured that either multiple messages will land at the same server in a load-balanced pool or the server processing the message will accurately access the state remembered from the previous messages (and behave as if it were the same server).
CLOSING. The last message (or messages flowing together in the same direction) gets sent.

Each of these stages of communication offers challenges, especially the initiation and closing stages.

THE INITIATION-STAGE AMBIGUITY

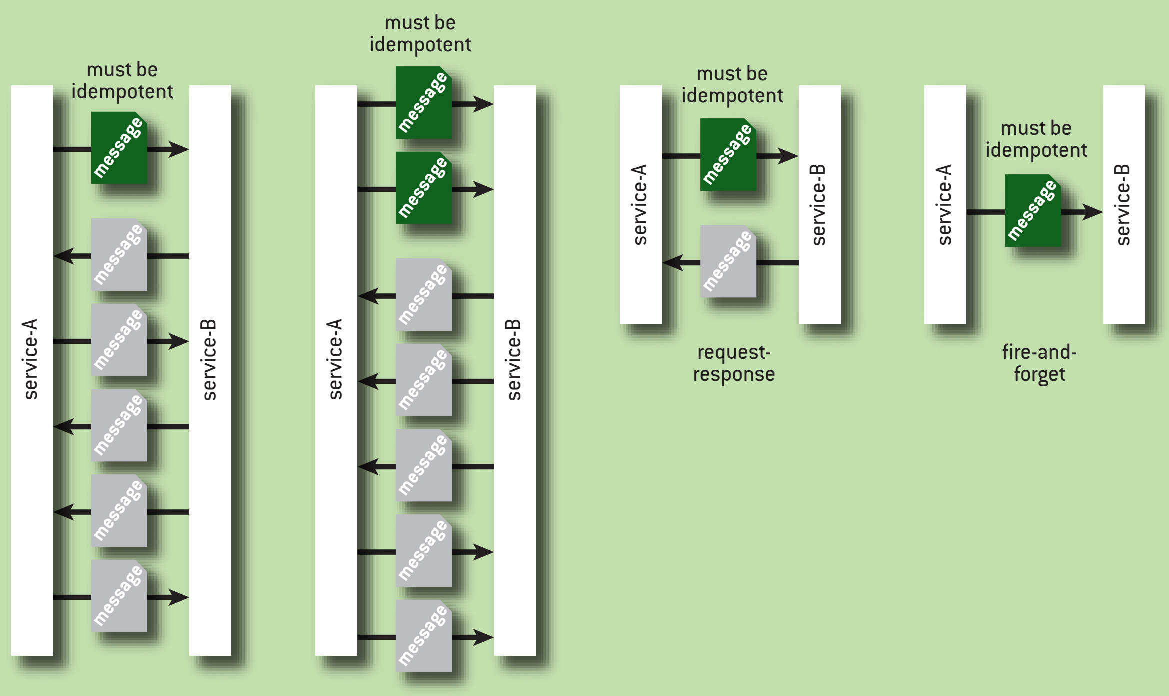
When the first message in a dialog is sent, it may or may not need retrying. In a load-balanced server environment, the two retries may land at different back-end servers and be processed independently.

From the sender's perspective, the application tosses a message into the plumbing with some name that hopefully will get it to the desired partner. Until you hear a response, you can't tell if the message was received, the message was lost, the response was lost, or the work is still in progress. The request must be idempotent (see figure 8).

The first message sent to a service must be idempotent because it may be retried in order to cope with transmission failures. Subsequent messages can count on some plumbing to help (provided that the application runs on top of the plumbing). After the first message, the plumbing can know enough about the destination of the message (in a scalable system) to perform automatic duplicate

FIGURE 8

The First Message Sent to a Service Must Be Idempotent



elimination. During the processing of the first message, the retries may land in different portions of the scalable service, and then automatic duplicate elimination is not possible.

I THINK MY EVIL TWIN GOT YOUR FIRST MESSAGE

Sometimes a server receives the first message (or messages) from a dialog, then a retry of the message (or sequence of messages) is rerouted to another server in the load-balanced pool. This can go awry in two ways:

• The messaging protocol is designed to keep the state in the target server and responds to the initiator with a more detailed address of the server within the pool for subsequent messages.

• The session state is kept separate from the load-balanced server, and subsequent messages in the protocol include session-identity information that allows the stateless load balancer to fetch the session state and keep going where the protocol left off.

These two approaches are equally challenged by a retry. The first attempt will not be correlated with the second attempt that happened as a result of the retry. This means the initiation messages in a dialog protocol *must be idempotent* (see figure 9).

When an initiating service sends a sequence of messages to a load-balanced service, the first message must be idempotent. Consider the following events:

1. A sequence of messages (1, 2, and 3) is sent to service Foo, and the load-balancer selects Foo A.
2. The work for the messages is performed at Foo A.

3. The answer is sent back indicating future messages should target Foo A as a resolved name. Unfortunately, the reply is lost by the flaky network transport.
4. A retry to service Foo happens, and the messages are sent to server Foo N.
5. The work for messages 1, 2, and 3 is performed at server Foo N.
6. The response is sent back to the initiator who now knows to keep chatting with Foo N. Somehow, we must ensure that the redundant work performed at Foo A is not a problem.

ACCURATELY ENDING THE INITIATION STAGE

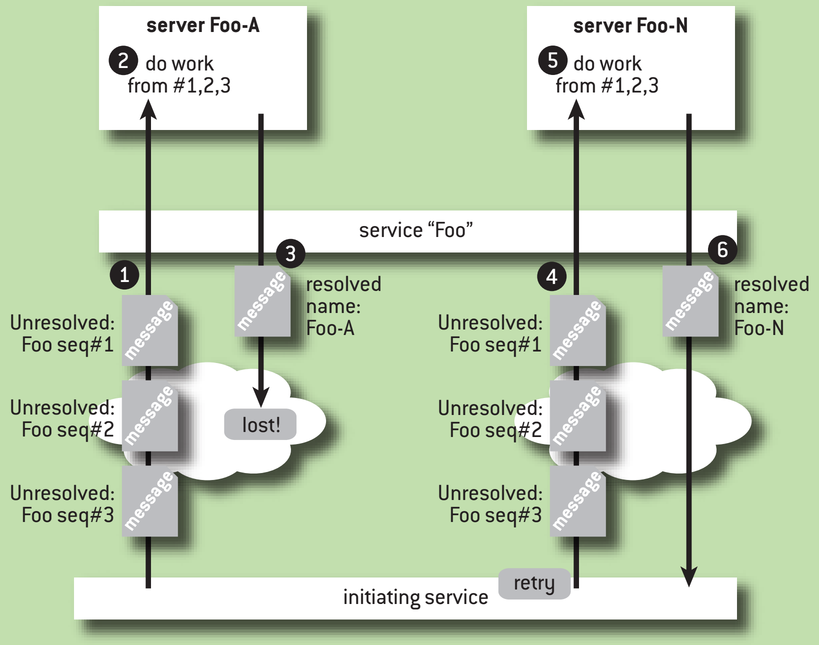
An application knows it has reached a specific partner when a message is returned from its local plumbing. If the other service has responded on the dialog, then there is a specific server or a bound session state for the dialog. While the application may not directly see the binding to the specific resources in the partner, they must have been connected by the time an application response is seen.

Prior to the application's receipt of a partner application's message from its local plumbing, any message that is sent may be rerouted to a new (and forgetful) implementation of the partner.

Only after you've heard from the application on the other side may you exit the initiation stage

FIGURE 9

The First Messages Must Be Idempotent



of the dialog (see figure 10). An application can see only what its plumbing shows it. When an app starts sending messages to its partner, it has the initiation-stage ambiguity and must ensure that all messages have a semantic for idempotent processing. When the local plumbing returns a message from the partner, the local application can be assured that the plumbing has resolved the initiation-stage ambiguity and now messages can be sent without having to ensure that they are idempotent.

GUARANTEEING IDEMPOTENCE OF THE FIRST MESSAGES

Everything you say as an application in a dialog before the first answer may be retried. That can cause a world of trouble if the early messages cause some serious (and non-idempotent) work to happen. What's an application developer to do?

There are three broad ways to make sure you don't have bugs in the initialization stage:

TRIVIAL WORK. You can simply send a message back and forth, which does nothing but establish the dialog to the specific partner in the scalable server. This is what TCP does with its SYN messages.

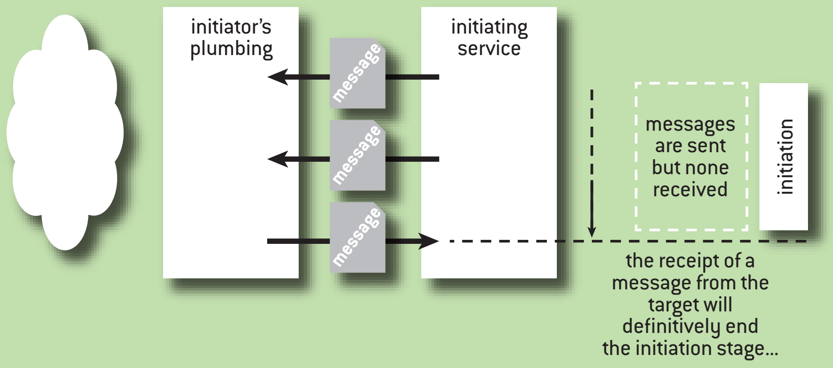
READ-ONLY WORK. The initiating application can read some stuff from the scalable server. This won't leave a mess if there are retries.

PENDING WORK. The initiating application can send a bunch of stuff, which the partner will accumulate. Only when subsequent round-trips have occurred (and you know which specific partner has been connected and the actual state of the dialog) will the accumulated state be permanently applied. If the server accumulates state and times out, then the accumulated state can be discarded without introducing bugs.

Using one of these three approaches, the application developer (and not the plumbing) will ensure that no bugs are lying in wait for a retry to a different back-end partner. To eliminate this risk completely, the plumbing can use TCP's trick and send a trivial round-trip set of messages (the SYN messages in TCP), which hooks up the partner without bothering the application layered on top. On the other hand, allowing the application to do useful work with the round-trip (e.g., read some data) is cool.

FIGURE 10

An Application Can Only See What Its Plumbing Shows It


initiator's plumbing
message
message
message
initiating service
messages are sent but none received
initiation
the receipt of a message from the target will definitively end the initiation stage...

THE CLOSING-STAGE AMBIGUITY

In any interaction, *the last message from one application service to another cannot be guaranteed*. The only way to know it was received is to send a message saying it was. That means it's no longer the last message.

Somehow, some way, the application must deal with the fact that the last message or messages sent in the same direction may simply go poof in the network. This may be in a simple request-response, or it may be in a complex full-duplex chatter. When the last messages are sent, it is just a matter of luck if they actually get delivered (see figure 11). In each interaction between two partners, the last messages sent in the same direction cannot be guaranteed. They may simply disappear.

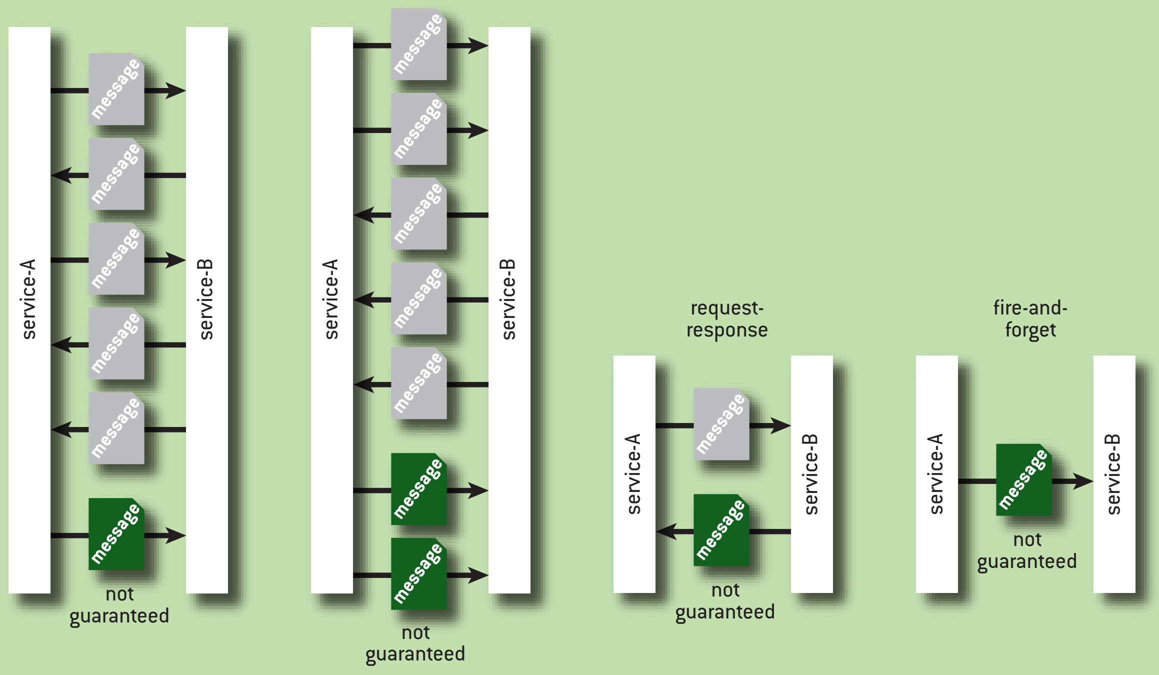
THE PENULTIMATE CONFUSION

The penultimate message can be guaranteed (by receiving the notification in the ultimate message). The ultimate message must be best effort. This complexity is typically part of designing an application protocol. The application must not really care if that last message is received because you can't really tell if it is.

LIVING WITH IDEMPOTENCE

As we've seen, most loosely coupled systems depend on the application designer to consider repeated processing of a request. The complications are made even worse in a scalable implementation of a service. Application designers must learn to live with the reality of idempotence in their daily life.

FIGURE 11 The Last Messages Sent in the Same Direction Cannot Be Guaranteed



CONCLUSION

Distributed systems can pose challenges to applications sending messages. The messaging transport can be downright mischievous. The target for the message may be an illusion of a partner implemented by a set of worker bees. These, in turn, may have challenges in their coordination of the state of your work. Also, the system you *think* you are talking to may, in fact, be subcontracting the work to other systems. This, too, can add to the confusion.

Sometimes an application has plumbing that captures its model of communicating partners, lifetime of the partners, scalability, failure management, and all the issues needed to have a great two-party dialog between communicating application components. Even in the presence of great supporting plumbing, there are still semantic challenges intrinsic to messaging.

This article has sketched a few principles used by grizzled old-timers to provide resilience even when "stuff happens." In most cases, these programming techniques are used as patches to applications when the rare anomalies occur in production. As a whole, they are not spoken about too often and rarely crop up during testing. They typically happen when the application is under its greatest stress (which may be the most costly time to realize you have a problem).

Some basic principles are:

- Every message may be retried and, hence, must be idempotent.
- Messages may be reordered.
- Your partner may experience amnesia as a result of failures, poorly managed durable state, or load-balancing switchover to its evil twin.
- Guaranteed delivery of the last message is impossible.

Keeping these principles in mind can lead to a more robust application.

While it is possible for plumbing or platforms to remove some of these concerns from the application, this can occur only when the communicating apps share common plumbing. The emergence of such common environments is not imminent (and may never happen). In the meantime, developers need to be thoughtful of these potential dysfunctions in erecting applications.

ACKNOWLEDGMENTS

Thank you to Erik Meijer and Jim Maurer for their commentary and editorial improvements.

REFERENCES

1.IBM. WebSphere MQ; http://www-01.ibm.com/software/integration/wmq/.

2. Tanenbaum, A. S. 2002. *Computer Networks,* 4th edition. Prentice Hall.

3. World Wide Web Consortium, Network Working Group. 1999. Hypertext Transfer Protocol – HTTP1.1; http://www.w3.org/Protocols/rfc2616/rfc2616.html.

4. Wolter, R. 2005. An introduction to SQL Server Service Broker; http://msdn.microsoft.com/en-us/library/ms345108(v=sql.90).aspx.

5. Transmission Control Protocol; http://www.ietf.org/rfc/rfc793.txt

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

PAT HELLAND has worked in distributed systems, transaction processing, databases, and similar areas

since 1978. For most of the 1980s, he was the chief architect of Tandem Computers' TMF (Transaction Monitoring Facility). With the exception of a two-year stint at Amazon, Helland worked at Microsoft from 1994 to 2011 where he was the architect for Microsoft Transaction Server and SQL Service Broker. He also contributed to Cosmos, a distributed computation and storage system that provides back-end support for Bing. He left Microsoft in 2011 when he moved to San Francisco to be nearer to family. That's left him more time to write articles for *ACM Queue*.