Life beyond Distributed Transactions

AN APOSTATE'S OPINION

PAT HELLAND

This is an updated and abbreviated version of a paper by the same name first published in CIDR (Conference on Innovative Database Research) 2007.

Transactions are amazingly powerful mechanisms, and I've spent the majority of my almost-40-year career working on them. In 1982, I first worked to provide transactions on the Tandem NonStop System. This system had a mean time between failures measured in years[4] and included a geographically distributed two-phase commit offering excellent availability for strongly consistent transactions.

New innovations, including Google's Spanner,[2] offer

strongly consistent transactional environments at extremely large scale with excellent availability. Building distributed transactions to support highly available applications is a great challenge that has inspired excellent innovation and great technology. Unfortunately, this is not broadly available to application developers.

In most distributed transaction systems, the failure of a single node causes transaction commit to stall. This in turn causes the application to get wedged. In such systems, the larger it gets, the more likely the system is going to be down. When flying an airplane that needs all of its engines to work, adding an engine reduces the availability of the airplane. Running a distributed transaction system over thousands of nodes is impractical without special mechanisms to tolerate outages. When application developers build systems using non-highly available distributed transactions, the solutions are brittle and must be discarded. Natural selection kicks in...

Instead, applications are built using techniques that do not provide transactional guarantees but still meet the needs of their business.

This article explores and names some of the practical approaches used in the implementation of large-scale mission-critical applications in a world that rejects distributed transactions. Topics include the management of fine-grained pieces of application data that may be repartitioned over time as the application grows. Design patterns support sending messages between these repartitionable pieces of data.

The goal here is to reduce the challenges faced by people handcrafting very large scalable applications.

Also, by observing these design patterns, the industry can perhaps work toward the creation of platforms to make it easier to develop scalable applications. Finally, while this article targets scalable homogeneous applications, these techniques are also very useful for supporting scalable heterogeneous applications such as support for mobile devices.

GOALS

This article focuses on how an application developer can build a successful scalable enterprise application when he or she has only a local database or transaction system available. Availability is not addressed, merely scale and correctness.

Discuss Scalable Applications

Most designers of scalable applications understand the business requirements. The problem is that the issues, concepts, and abstractions for the interaction of transactions and scalable systems have no names and are not crisply understood. They are inconsistently applied and sometimes come back to bite the designers. One goal of this article is to launch a discussion that can increase awareness of these concepts, leading toward a common set of terms and an agreed-upon approach to scalable programs.

Think about Almost-Infinite Scaling of Applications

The article presents an informal thought experiment on the impact of almost-infinite scaling. Let's assume the number of customers, purchasable entities, orders,

shipments, health-care patients, taxpayers, bank accounts, and all other business concepts manipulated by the app grows significantly over time. Typically, each individual thing doesn't get much larger; there are simply more and more of them. It really doesn't matter if CPU, DRAM, storage, or some other resource gets saturated first. At some point, the increase in demand leads to spreading what used to run on a single machine over a larger number of machines. This thought experiment makes us consider tens or hundreds of thousands of machines.

Almost-infinite scaling is a loose, imprecise, and deliberately amorphous way of motivating the need to be very clear about when and where you can *know* something fits on one machine and what to do if you cannot ensure it fits on one machine. Furthermore, you want to scale almost linearly with the data and computation load. Of course, scaling at an N-log-N pace with a big log would be great.

Describe a Few Common Patterns for Scalable Apps

What are the impacts of almost-infinite scaling on the business logic? I am asserting that scaling implies using a new abstraction called an *entity* as you write your program. An entity lives on a single machine at a time, and the application can manipulate only one entity at a time. A consequence of almost-infinite scaling is that this programmatic abstraction must be exposed to the developer of the business logic.

By naming and discussing this as-yet-unnamed concept, we can perhaps agree on a consistent programmatic approach and a consistent understanding of the issues involved in building scalable systems.

Furthermore, the use of entities has implications for the messaging patterns used to connect them. This leads to the creation of state machines that cope with the message-delivery inconsistencies foisted upon innocent application developers attempting to build scalable solutions to business problems.

SOME ASSUMPTIONS

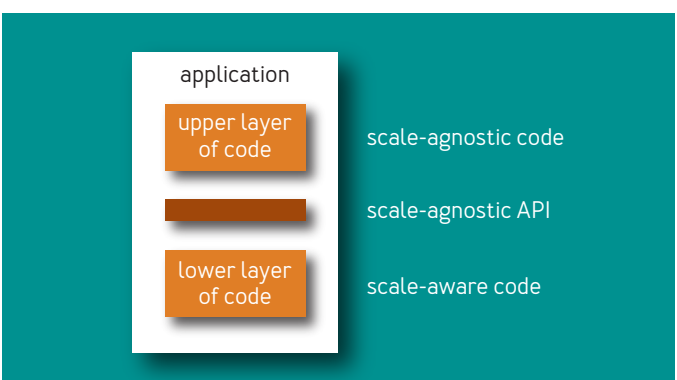
Consider the following three assumptions, which are asserted and not justified. Assume these are true based on experience.

Layers of the Application and Scale Agnosticism

Let's begin by presuming that each scalable application has at least two layers, as shown in figure 1. These layers differ in the perception of scaling. They may have other differences, but these are not relevant to this discussion.

The lower layer of the application understands that more computers get added to make the system scale.

FIGURE 1: **TWO-LAYERED APPLICATION**



In addition to other work, it manages the mapping of the upper layer's code to the physical machines and their locations. The lower layer is *scale-aware* in that it understands this mapping. I presume that the lower layer provides a *scale-agnostic programming abstraction* to the upper layer. There are many examples of scale-agnostic programming abstractions, including MapReduce.[3]

Using this scale-agnostic programming abstraction, the upper layer of the application code is written without worrying about scaling issues. By sticking to the scale-agnostic programmatic abstraction, you can write application code that is not worried about the changes happening when it is deployed over an ever-increasing load.

Over time, the lower layer of these applications may evolve to become a new platform or middleware that simplifies the creation of scale-agnostic APIs.

Transactional Scopes

Lots of academic work has been done on the notion of providing strongly consistent transactions over distributed systems. This includes 2PC (two-phase commit),[1] Paxos,[5] and recently Raft.[6] Classic 2PC will block when a machine fails unless the coordinator and participants in the transaction are fault tolerant in their own right, such as the Tandem NonStop System. Paxos and Raft do not block with node failures but do extra work coordinating, much like Tandem's system.

These algorithms can be described as providing *strongly consistent transactions over distributed systems*. Their goal is to allow arbitrary atomic updates to data spread over

many machines. Updates exist in a single transactional scope spanning many machines.

Unfortunately, in many circumstances this is not an option for an application developer. Applications may need to span trust boundaries, different platforms, and different operational and deployment zones. What happens when you "just say no" to distributed transactions?

Even today, 10 years after this paper was first written, real system developers rarely try to achieve strongly consistent transactions over more than just a few computers. Instead, they assume multiple separate transaction scopes. Each computer is a separate scope with local transactions inside.

Most Applications Use At-Least-Once Messaging

TCP/IP is great if you are a short-lived Unix-style process, but consider the dilemma faced by an application developer whose job is to process a message and modify some durable data represented in a database. The message is consumed and not yet acknowledged. The database is updated and then the message is acknowledged. In a failure, this is restarted and the message is processed again.

The dilemma derives from the fact that the message delivery is not directly coupled to the update of the durable data other than through application action. While it is possible to couple the consumption of messages to the update of the durable data, this is not commonly available. The absence of this coupling leads to failure windows where the message is delivered more than once. Rather than lose messages, the message plumbing delivers them at least once.

A consequence of this behavior is that the application must tolerate message retries and out-of-order delivery.

OPINIONS TO BE JUSTIFIED

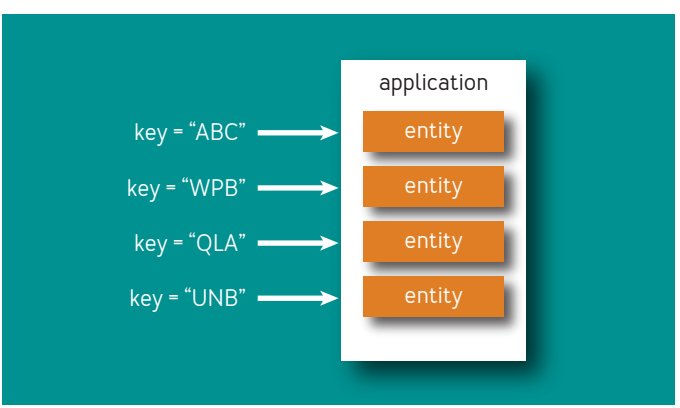
The nice thing about writing an opinion piece is that you can express wild opinions. Here are a few that this article tries to justify.

Scalable Apps Use Uniquely Identified Entities

This article argues that the upper-layer code for each application must manipulate a single collection of data called an *entity*. There are no restrictions on the size of a single entity except that it must live within a single transactional scope (i.e., one machine).

Each entity has a unique identifier or key, as shown in figure 2. An entity key may be of any shape, form, or flavor. Somehow, it must uniquely identify exactly one entity and the data it contains.

FIGURE 2: **DATA FOR AN APPLICATION COMPRISES MANY ENTITIES**



There are no restrictions on the representations of the entity. It may be represented as SQL records, XML, JSON, files, or anything else. One possible representation would be a collection of SQL records, potentially across many tables, whose primary key has the entity key as its prefix.

Entities represent disjoint sets of data. Each datum resides in exactly one entity.

An application consists of many entities. For example, an order-processing application encapsulates many orders, each of which is identified by a unique Order-ID. To be a scalable order-processing application, data from one order must be disjoint from data for other orders.

Atomic Transactions Cannot Span Entities

Each computer is assumed to be a separate transactional scope. Later this article presents the argument that atomic transactions cannot span entities. The programmer must always stick to the data contained inside a single entity for each transaction.

From the programmer's perspective, *the uniquely identified entity is the transactional scope*. This concept has a powerful impact on the behavior of applications designed for scaling. One implication to be explored is that alternate indices cannot be kept transactionally consistent when designing for almost-infinite scaling.

Messages Are Addressed to Entities

Most messaging systems do not consider the partitioning key for the data but rather target a queue that is consumed by a stateless process. Standard practice is to include some data in the message that informs the stateless

application code of where to get the data it needs. This is the entity key. The data for the entity is fetched from some database or other durable store by the application.

A couple of interesting trends are happening. First, the size of the *set* of entities is growing larger than will fit on a single computer. Each individual entity usually fits in one computer, but the set of them does not. Increasingly, the stateless application is routing to fetch the entity based on some partitioning scheme.

Second, the fetching and partitioning scheme is being separated into the lower layers of the application. This is deliberately isolated from the upper layers responsible for the business logic.

This pattern effectively targets the entity by routing using the entity key. Both the stateless Unix-style process and the lower layers of the application are simply part of the implementation of the scale-agnostic API provided for the business logic. The upper-layer scale-agnostic business logic simply addresses the message to the entity key that identifies the durable state known as the entity.

Entities Manage Per-Partner State (Activities)

Scale-agnostic messaging is effectively entity-to-entity messaging. The sending entity is manifest by its durable state and is identified by its entity key. It sends a message to another entity and identifies it by its entity key. The recipient entity consists of both scale-agnostic upper-layer business logic and the durable data representing its state. This is identified by its entity key.

Recall the assumption that messages are delivered at least once. The recipient entity may be assailed with

redundant messages that must be ignored. In practice, messages fall into two categories: those that affect the state of the entity and those that do not. Messages that don't affect the entity's state are easy—they are trivially idempotent. Messages that change the state require more care.

To ensure idempotence (i.e., guarantee that the processing of retried messages is harmless), the recipient entity is typically designed to remember that the message has been processed. Once it has been successfully processed, repeated messages will typically generate another response matching the behavior of the first message.

The knowledge of the received message creates state that is wrapped up on a per-partner basis. The important observation is that the state is organized on a per-partner basis and each partner is an entity.

The term *activity* is applied to the state that manages the per-partner messaging on each side of a two-party relationship. Each activity lives in exactly one entity. An entity will have an activity for each partner entity.

In addition to managing messaging melees, activities are used to manage loosely coupled agreements. In a world where atomic transactions are not a possibility, tentative operations are used to negotiate a shared outcome. These are performed between entities and are managed by activities.

Building workflows to reach agreement is fraught with challenges that are well documented elsewhere.[7] This article does not assert that activities solve these challenges, but rather that they give a foundation for

storing the state needed to solve them. Almost-infinite scaling leads to surprisingly fine-grained workflow-style solutions. The participants are entities, and each entity manages its workflow using specific knowledge about the other entities involved. That two-party knowledge maintained inside an entity is called an activity.

Examples of activities are sometimes subtle. An order application sends messages to the shipping application. It includes the shipping ID and the sending order ID. The message type may be used to stimulate the state changes in the shipping application to record that the specified order is ready to ship. Frequently, implementers don't design for retries until a bug appears. Rarely, but occasionally, the application designers think about and plan for activities.

ENTITIES

This section examines the nature of entities in greater depth.

Disjoint Transactional Scopes

Each entity is defined as a collection of data with a unique key known to live within a single transactional scope. Atomic transactions may always be done *within a single entity.*

Uniquely Keyed Entities

Code for the upper layer of an application is naturally designed around collections of data with a unique key. Customer IDs, Social Security numbers, product SKUs, and other unique identifiers can be seen within applications.

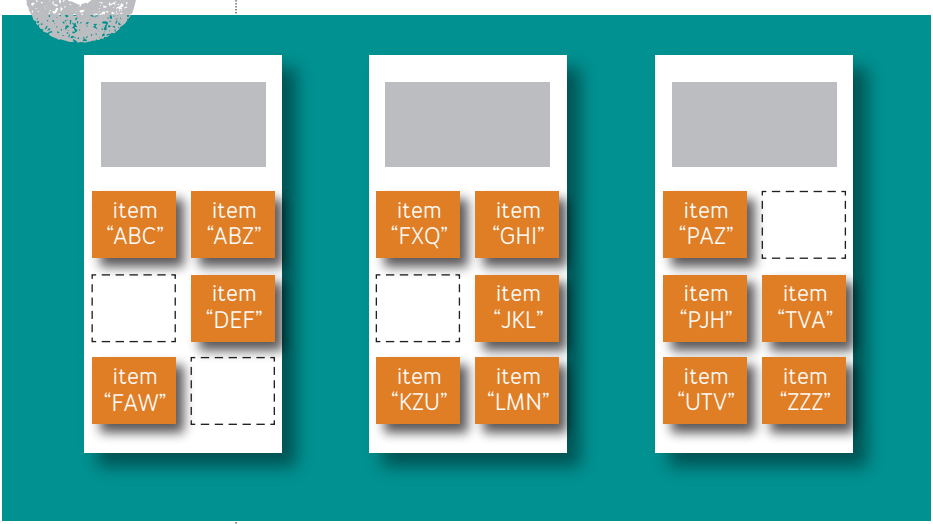
They are used as keys to locate the applications' data. Guarantees of transactional atomicity come only within an entity identified by a unique key.

Repartitioning and Entities

One of the assumptions previously stated is that the emerging upper layer is scale agnostic, and the lower layer decides how the deployment evolves as its scale changes. The location of a specific entity is likely to change as the deployment evolves. The upper layer of the application cannot make assumptions about the location of the entity because that would not be scale agnostic.

As shown in figure 3, entities are spread across transactional scopes using either hashing or key-range partitioning.

FIGURE 3: **ENTITIES SPREAD ACROSS DIFFERENT TRANSACTIONAL SCOPES**



Atomic Transactions and Entities

In scalable systems, *you can't assume transactions for updates across these different entities.* Each entity has a unique key, and each entity is easily placed into one transactional scope. Recall the premise that almost-infinite scaling causes *the number of entities* inexorably to increase, but *the size of the individual entity* remains small enough to fit in a transactional scope (i.e., one computer).

How can you know that two separate entities are guaranteed to be within the same transactional scope and, hence, atomically updatable? You know only when a single unique key unifies both. Now it is really one entity!

If hashing is used for partitioning by entity key, there's no telling when two entities with different keys land in the same box. If key-range partitioning is used for the entity keys, most of the time the adjacent key values reside on the same machine. Once in a while you will get unlucky and your neighbor will be on another machine.

A simple test case that counts on atomicity with a neighbor in a key-range partitioning will usually succeed. Later, when redeployment moves the entities across machines, the latent bug emerges; the updates are no longer atomic. You can never count on different entity-key values residing in the same place.

Put more simply, the lower layer of the application will ensure that each entity key (and its entity) resides on a single machine. Different entities may be anywhere.

A scale-agnostic programming abstraction must have the notion of entity as the boundary of atomicity. Understanding entities, the use of the entity key, and the clear commitment to a lack of atomicity across entities is

essential to scale-agnostic programming.

Large-scale applications implicitly do this in the industry today; there just isn't a name for the concept of an entity. From an upper-layer app's perspective, it must assume that the entity is the scope of transactions. Assuming more will break when the deployment changes.

Considering Alternate Indices

We are accustomed to the ability to address data with multiple keys or indices. For example, sometimes a customer is referenced by Social Security number, sometimes by credit card number, and sometimes by street address. Assuming extreme amounts of scaling, these indices cannot reside on the same machine or in a single large cluster. *The data about a single customer cannot be known to reside within a single transactional scope.* The entity itself resides in a single transactional scope. The challenge is that the copies of the information used to create an alternate index must be assumed to reside in a different transactional scope.

Consider guaranteeing that the alternate index resides in the same transactional scope. When almost-infinite scaling kicks in, the set of entities is smeared across gigantic numbers of machines. The primary index and alternate index information must reside within the same transactional scope. The only way to ensure this is to locate the alternate index using the primary index. That takes you to the same transactional scope. If you start without the primary index and have to search all of the transactional scopes, each alternate index lookup must examine an almost-infinite number of scopes as it looks for

the match to the alternate key. This will eventually become untenable.

The only logical alternative is to do a two-step lookup. First, look up the alternate key, which yields the entity key. Second, access the entity using the entity key. This is very much like inside a relational database as it uses two steps to access a record via an alternate key. But the premise of almost-infinite scaling means the two indices (primary and alternate) cannot be known to reside in the same transactional scope. See figure 4.

The scale-agnostic application program can't atomically update an entity and its alternate index. The upper-layer scale-agnostic application must be designed to understand that alternate indices may be out of sync with the entity accessed with its primary index (i.e., entity key). As shown in figure 4, different keys (primary entity key versus alternate keys) cannot be collocated or updated atomically.

What in the past has been managed automatically as alternate indices must now be managed manually by the application. Workflow-style updates via asynchronous messaging are all that is left. When you read data from alternate indices, you must understand that it is potentially out of sync with the entity itself. Alternate indices are now harder. This is a fact of life in the big cruel world of huge systems.

MESSAGING ACROSS ENTITIES

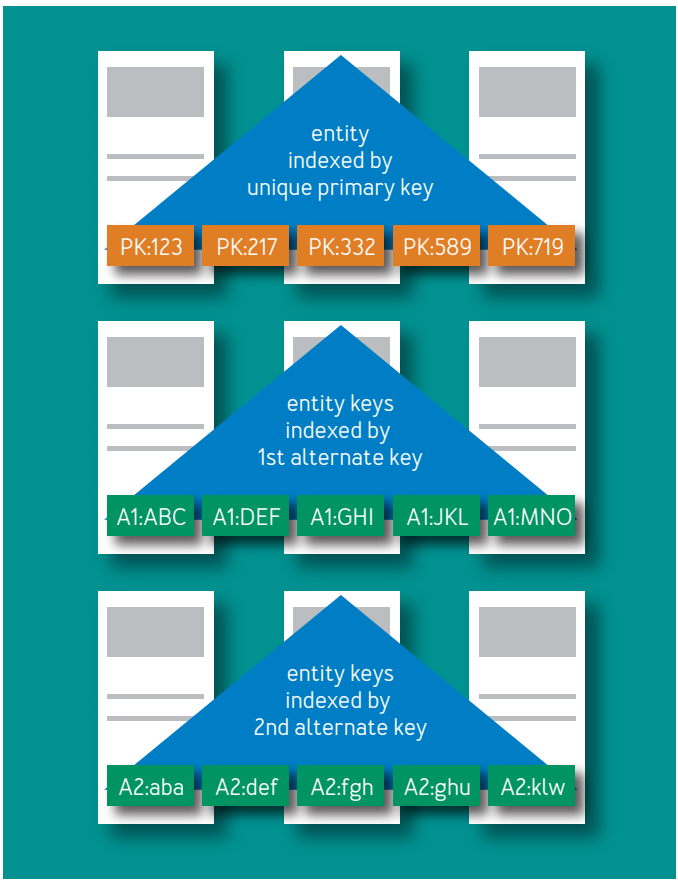
This section considers connecting independent entities using messages. It examines naming, transactions and messages, message-delivery semantics, and the impact of repartitioning entities.

Messages to Communicate across Entities

If you can't update the data across two entities in the same

4

FIGURE 4: **UNDER SCALE, ALTERNATE INDEX ENTRIES LAND ELSEWHERE**



transaction, you need a mechanism to update the data in different transactions. The connection between the entities is via a message.

Asynchronous with Respect to Sending Transactions

Since messages are across entities, the data associated

with the decision to send the message is in one entity, and the destination of the message is in another entity. By the definition of an entity, these entities cannot be atomically updated. Messages cannot be atomically sent and received across these different entities.

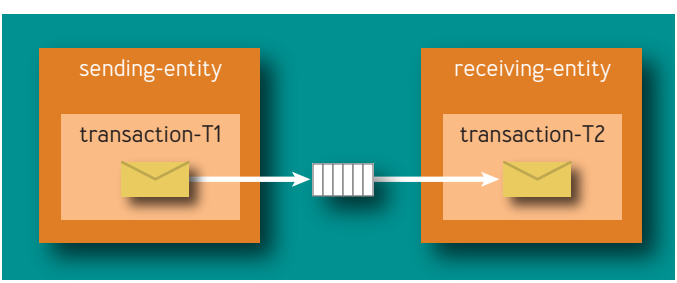
It would be horribly complex for an application developer to send a message while working on a transaction, have the message sent, and then have the transaction abort. This would mean you had no memory of causing something to happen and yet it did happen. For this reason, transactional enqueuing of messages is *de rigueur*. See figure 5.

If the message cannot be seen at the destination until *after* the sending transaction commits, the message is asynchronous with respect to the sending transaction. Each entity advances to a new state with a transaction. Messages are the stimuli coming from one transaction and arriving at a new entity causing transactions.

Naming the Destination of Messages

Consider the programming of the scale-agnostic part of

FIGURE 5: **MESSAGES FLOW FROM ONE ENTITY TO ANOTHER**



an application, as one entity wants to send a message to another entity. The location of the destination entity is not known to the scale-agnostic code. The entity key is.

It falls on the scale-aware part of the application to correlate the entity key to the location of the entity.

Repartitioning and Message Delivery

When the scale-agnostic part of the application sends a message, the lower-level scale-aware portion hunts down the destination and delivers the message at least once.

As the system scales, entities move. This is commonly called *repartitioning*. The location of the entity and, hence, the destination of the message may be in flux. Sometimes messages will chase to the old location only to find out the pesky entity has been sent elsewhere. Now the message will have to follow.

As entities move, the clarity of a first-in, first-out queue between the sender and the destination is occasionally disrupted. Messages are repeated. Later messages arrive before earlier ones. Life gets messier.

For these reasons, scale-agnostic applications are evolving to support idempotent processing of all application-visible messaging. This implies reordering in message delivery, too.

ACTIVITIES: COPING WITH MESSY MESSAGES

This section discusses ways of coping with the challenges of message retries and reordering. It introduces the notion of an activity as the local information needed to manage a relationship with a partner entity.

Retries and Idempotence

Since any message may be delivered multiple times, the application needs a discipline to cope with repeated messages. While it is possible to build low-level support for the elimination of duplicate messages, in an almost-infinite scaling environment the low-level support would need to know about entities. The knowledge of which messages have been delivered to the entity must travel with the entity when it moves because of repartitioning. In practice, the low-level management of this knowledge rarely occurs; messages may be delivered more than once.

Typically, the scale-agnostic (higher-level) portion of the application must implement mechanisms to ensure that the incoming message is idempotent. This is not essential to the nature of the problem. Duplicate elimination could certainly be built into the scale-aware parts of the application. This is not yet available. Hence, let's consider what the poor developer of the scale-agnostic application must implement.

Defining Idempotence of Substantive Behavior

The processing of a message is idempotent if a subsequent execution of the processing does not perform a *substantive change* to the entity. This is an amorphous definition that leaves open to the application the specification of what is and what is not substantive.

If a message does not change the invoked entity but only reads information, its processing is idempotent. This is true even if a log record describing the read is written. The log record is not substantive to the behavior of the entity. The definition of what is and what is not substantive is application specific.

Natural Idempotence

To accomplish idempotence, it is essential that the message does not cause substantive side effects. Some messages provoke no substantive work any time they are processed. These are *naturally* idempotent.

A message that only reads some data from an entity is naturally idempotent. What if the processing of a message does change the entity but not in a substantive way? Those, too, would be naturally idempotent.

Now it gets harder. The work implied by some messages actually causes substantive changes. These messages are not naturally idempotent. The application must include mechanisms to ensure that these messages, too, are idempotent. This means remembering in some fashion that the message has been processed so that subsequent attempts make no substantive change.

The next section considers the processing of messages that are not naturally idempotent.

Remembering Messages as State

To ensure the idempotent processing of messages that are not naturally idempotent, the entity must remember they have been processed. This knowledge is state. The state accumulates as messages are processed.

In addition, if a reply is required, the same reply must be returned. After all, you don't know if the original sender has received the reply.

Managing State for Each Partner

To track relationships and the messages received, each entity within the scale-agnostic application must somehow

remember state information about its partners. It must capture this state on a partner-by-partner basis. Let's name this state an *activity*. See figure 6. Each entity may have many activities if it interacts with many other entities. Activities track the interactions with each partner.

Each entity consists of a set of activities and, perhaps, some other data that spans the activities.

Consider the processing of an order consisting of many items for purchase. Reserving inventory for shipment of each separate item will be a separate activity. There will be an entity for the order and separate entities for each item managed by the warehouse. Transactions cannot be assumed across these entities.

Within the order, each inventory item will be separately managed. The messaging protocol must be separately managed. The per-inventory-item data contained within the order entity is an activity. While it is not named as such, this pattern frequently exists in large-scale apps.

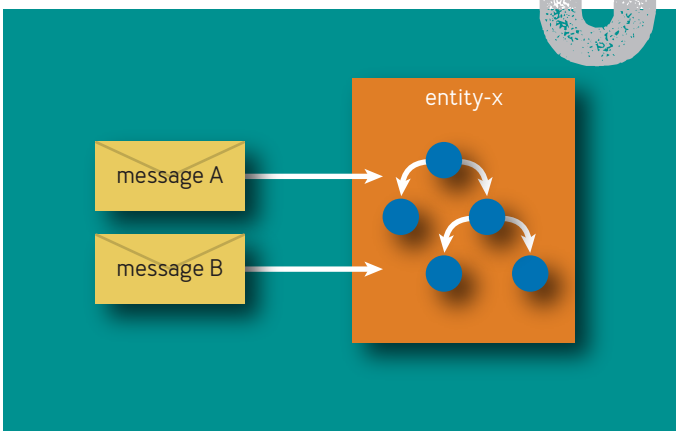
In an almost-infinitely scaled application, you need to be very clear about relationships. You can't just do a query to figure out what is related. Everything must be formally knit together using a web of two-party relationships. The knitting is done with the entity keys. Because the partner is some distance away, you have to formally manage your understanding of the partner's state as new knowledge when the partner arrives. The local information known about a distant partner is referred to as an activity. See figure 7.

Ensuring At-Most-Once Acceptance via Activities

Processing messages that are not naturally idempotent requires ensuring that each message is processed at most once (i.e., the substantive impact of the message must happen at most once). To do this, some unique characteristic of the message must be remembered to

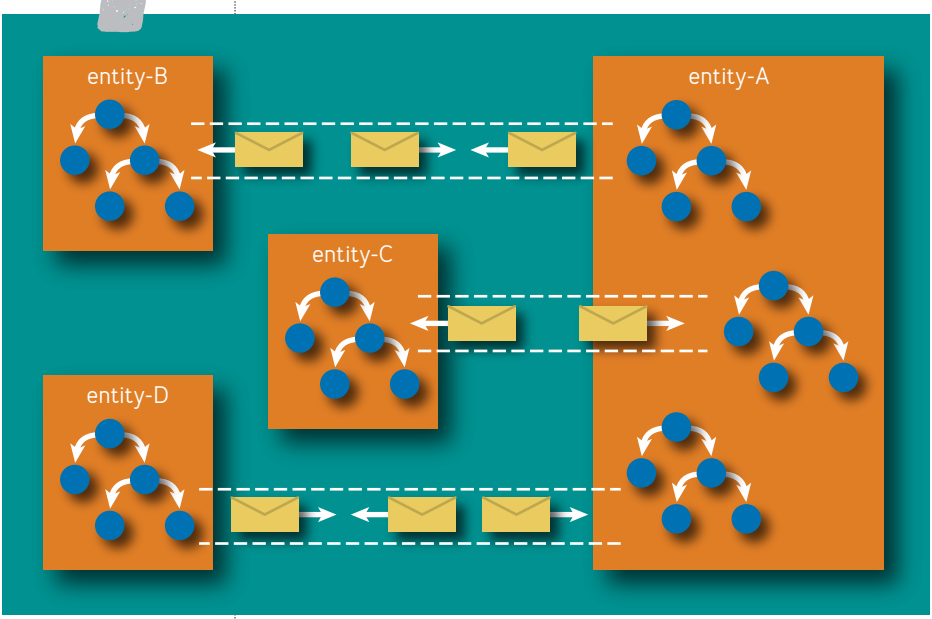
6

FIGURE 6: **ACTIVITIES ARE WHAT AN ENTITY REMEMBERS**



7

FIGURE 7: **EXAMPLE OF ONE ACTIVITY PER PARTNER**



ensure it will not be processed more than once.

The entity must durably remember the transition from a message being OK to process into the state where the message will not have substantive impact.

Typically, an entity uses its activities to implement this state management on a partner-by-partner basis. This is essential because sometimes an entity supports many different partners, and each will pass through a pattern of messages associated with that relationship. The per-partner collection of state makes that possible.

ACTIVITIES: COPING WITHOUT ATOMICITY

This section addresses how wildly scalable systems make decisions without distributed transactions. Managing distributed agreement is hard work. In an almost-infinitely scalable environment, the representation of uncertainty must be done in a fine-grained fashion that is oriented around per-partner relationships. This data is managed within entities using the notion of an activity.

Uncertainty at a Distance

The absence of distributed transactions means the acceptance of uncertainty when attempting to come to decisions across different entities. It is unavoidable that decisions across distributed systems involve accepting uncertainty for a while. When distributed transactions can be used, that uncertainty is manifested in the locks held on data and is managed by the transaction manager. In a system that cannot count on distributed transactions, the management of uncertainty must be implemented in the business logic. The uncertainty of the outcome is held in

the business semantics rather than in the record lock. This is simply workflow. It's not magic. You can't use distributed transactions, so you use workflow. The assumptions that led to entities and messages now lead to the conclusion that the scale-agnostic application must manage uncertainty itself using workflow. This is needed to reach agreement across multiple entities.

Think about the style of interactions common across businesses. Contracts between businesses include time commitments, cancellation clauses, reserved resources, and much more. Uncertainty is wrapped up in the behavior of the business functionality. While more complicated to implement than using distributed transactions, it is how the real world works...

Again, this is simply an argument for workflow.

Activities and the Management of Uncertainty

Entities sometimes accept uncertainty as they interact with other entities. This uncertainty must be managed on a partner-by-partner basis and can be visualized as being reified in the activity state for the partner. Many times, uncertainty is represented by relationship. It is necessary to track it by partner. The activity tracks each partner as it advances into a new state.

Performing Tentative Business Operations

To reach an agreement across entities, one entity asks another to accept uncertainty. One entity sends another a request that may be canceled later. This is called a *tentative operation*. At the end of this step, one entity agrees to abide by the wishes of the other.

If an ordering system reserves inventory from a warehouse, the warehouse allocates the inventory without knowing if it will be used. That is accepting uncertainty. Later, the warehouse finds out if the reserved inventory will be needed. This resolves the uncertainty.

The warehouse inventory manager must keep relationship data for each order encumbering its items. As it connects items and orders, these will be organized by item. Each item keeps information about outstanding orders for that item. Each activity within the item (one per order) manages the uncertainty of the order.

Tentative Operations, Confirmation, and Cancellation

Essential to a tentative operation is the right to cancel. If the requesting entity decides not to move forward, it issues a *cancellation operation*. If it decides to move ahead, it issues a *confirmation operation*.

When an entity agrees to perform a tentative operation, it agrees to let another entity decide the outcome. It accepts uncertainty, which adds to its confusion. As confirmations and cancellations arrive, the uncertainty decreases, reducing confusion. It is normal to proceed through life with ever increasing and decreasing uncertainty as old problems get resolved and new ones arrive in your lap.

Again, this is simply workflow, but it is fine-grained workflow with entities as the participants.

Uncertainty and Almost-Infinite Scaling

The management of uncertainty usually revolves around two-party agreements. There may be multiple two-party agreements. These are knit together as a web of fine-grained two-party agreements using entity keys as the links and activities to track the known state of a distant partner.

Consider a house purchase and the relationships with the escrow company. The buyer enters into an agreement of trust with the escrow company, as do the seller, mortgage company, and all other parties involved in the transaction.

When you go to sign papers to buy a house, you do not know the outcome of the deal. You accept that, until escrow closes, you are uncertain. The only party with control over the decision-making is the escrow company.

This is a hub-and-spoke collection of two-party relationships that are used to get a large set of parties to agree without use of distributed transactions.

When considering almost-infinite scaling, it is interesting to think about two-party relationships. By building up from two-party tentative/cancel/confirm (just like traditional workflow), you can see the basis for achieving distributed agreement. Just as in the escrow company (see sidebar), many entities may participate in an agreement through composition. Because the relationships are two-party, the simple concept of an activity as "stuff I remember about that partner" becomes a basis for managing enormous systems—even when the data is stored in entities and you don't know where the entity lives. You must assume it is far away. Still, it can be programmed in a scale-agnostic way. Real-world almost-infinite scale applications would love the luxury of a global transactional scope. Unfortunately, this is not readily available to most of us without introducing fragility when a system fails.

Instead, the management of the uncertainty of the tentative work is passed off into the hands of the developer of the scale-agnostic application. It must be handled as reserved inventory, allocations against credit lines, and other application-specific concepts.

CONCLUSIONS

As usual, the computer industry is in flux.

Today, new design pressures are being foisted onto programmers who simply want to solve business problems. Their realities are taking them into a world of almost-infinite scaling and forcing them into design problems largely unrelated to the real business at hand. This is true today, and it was true when this article was first published in 2007. Unfortunately, programmers striving to solve business goals such as e-commerce, supply-chain management, financial, and health-care applications increasingly need to think about scaling without distributed transactions. Most developers simply don't have access to robust systems offering scalable distributed transactions.

We are at a juncture where the patterns for building these applications can be seen, but no one is yet applying these patterns consistently. This article argues that these nascent patterns can be applied more consistently in the handcrafted development of applications designed for almost-infinite scaling.

This article has introduced and named a couple of formalisms emerging in large-scale applications:
• *Entities* are collections of named (keyed) data that may be atomically updated within the entity but never atomically updated across entities. • *Activities* consist of the collection of state within the entities used to manage messaging relationships with a single partner entity. Workflow to reach decisions functions within activities within entities. It is the fine-grained nature of workflow that is surprising when looking at almost-infinite scaling.

Many applications are implicitly being designed with both entities and activities today. They are simply not formalized, nor are they used consistently. Where the use is inconsistent, bugs are found and eventually patched. By discussing and consistently using these patterns, better large-scale applications can be built and, as an industry, we can get closer to building solutions that allow business-logic programmers to concentrate on the business problems rather than the problems of scale.

References

1. Bernstein, P. A., Hadzilacos, V., Goodman, N. 1987. *Concurrency Control and Recovery in Database Systems*. Boston, MA: Addison-Wesley.
2. Corbett, J. C., et al. 2012. Spanner: Google's globally distributed database. Tenth Usenix Symposium on Operating Systems Design and Implementation (OSDI).
3. Dean, J., Ghemawat, S. 2004. MapReduce: simplified data processing on large clusters. Sixth Symposium on Operating Systems Design and Implementation (OSDI).
4. Gray, J. 1990. A census of Tandem System availability between 1985 and 1990. *IEEE Transactions on Reliability* 39(4): 409-418.
5. Lamport, L. 1998. The part-time parliament. *ACM Transactions on Computer Systems* 16(2): 133-169.
6. Ongaro, D., Ousterhout, J. 2014. In search of an understandable consensus algorithm (extended version); https://raft.github.io/raft.pdf.
7. Wachter, H., Reuter, A. 1992. The ConTract Model. In *Database Transaction Models for Advanced Applications*: 219-263. San Francisco, CA: Morgan Kaufmann.

Related articles

A Conversation with Bruce Lindsay
Designing for failure may be the key to success
http://queue.acm.org/detail.cfm?id=1036486

Condos and Clouds
Constraints in an environment empower the services.
By Pat Helland
http://queue.acm.org/detail.cfm?id=2398392

Microsoft's Protocol Documentation Program: Interoperability Testing at Scale
A discussion with Nico Kicillof, Wolfgang Grieskamp, and Bob Binder
http://queue.acm.org/detail.cfm?id=1996412

Pat Helland *has been implementing transaction systems, databases, application platforms, distributed systems, fault-tolerant systems, and messaging systems since 1978. For recreation, he occasionally writes technical papers. He currently works at Salesforce.*