Life beyond Distributed Transactions: an Apostate's Opinion

Position Paper

Pat Helland

Amazon.Com
705 Fifth Ave South
Seattle, WA 98104
USA
PHelland@Amazon.com

The positions expressed in this paper are personal opinions and do not in any way reflect the positions of my employer Amazon.com.

ABSTRACT

Many decades of work have been invested in the area of distributed transactions including protocols such as 2PC, Paxos, and various approaches to quorum. These protocols provide the application programmer a façade of global serializability. Personally, I have invested a non-trivial portion of my career as a strong advocate for the implementation and use of platforms providing guarantees of global serializability.

My experience over the last decade has led me to liken these platforms to the Maginot Line[1]. In general, application developers simply do not implement large scalable applications assuming distributed transactions. When they attempt to use distributed transactions, the projects founder because the performance costs and fragility make them impractical. Natural selection kicks in…

Instead, applications are built using different techniques which do not provide the same transactional guarantees but still meet the needs of their businesses.

This paper explores and names some of the practical approaches used in the implementations of large-scale mission-critical applications in a world which rejects distributed transactions. We discuss the management of fine-grained pieces of application data which may be repartitioned over time as the application grows. We also discuss the design patterns used in sending messages between these repartitionable pieces of data.

The reason for starting this discussion is to raise awareness of new patterns for two reasons. First, it is my belief that this awareness can ease the challenges of people hand-crafting very large scalable applications. Second, by observing the patterns, hopefully the industry can work towards the creation of platforms that make it easier to build these very large applications.

1. INTRODUCTION

Let's examine some goals for this paper, some assumptions that I am making for this discussion, and then some opinions derived from the assumptions. While I am keenly interested in high availability, this paper will ignore that issue and focus on scalability alone. In particular, we focus on the implications that fall out of assuming we cannot have large-scale distributed transactions.

Goals

This paper has three broad goals:

- Discuss Scalable Applications
  Many of the requirements for the design of scalable systems are understood implicitly by many application designers who build large systems.

[1] The Maginot Line was a huge fortress that ran the length of the Franco-German border and was constructed at great expense between World War I and World War II. It successfully kept the German army from directly crossing the border between France and Germany. It was quickly bypassed by the Germans in 1940 who invaded through Belgium.

This article is published under a Creative Commons License Agreement (http://creativecommons.org/licenses/by/2.5/).
You may copy, distribute, display, and perform the work, make derivative works and make commercial use of the work, but you must attribute the work to the author and CIDR 2007.

3rd Biennial Conference on Innovative DataSystems Research (CIDR) January 7-10, Asilomar, California USA.

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. When they get applied, they are inconsistently applied and sometimes come back to bite us. One goal of this paper is to launch a discussion which can increase awareness of these concepts and, hopefully, drive towards a common set of terms and an agreed approach to scalable programs.

This paper attempts to name and formalize some abstractions implicitly in use for years to implement scalable systems.

- Think about Almost-Infinite Scaling of Applications
  To frame the discussion on scaling, this paper presents an informal thought experiment on the impact of almost-infinite scaling. I assume the number of customers, purchasable entities, orders, shipments, health-care-patients, taxpayers, bank accounts, and all other business concepts manipulated by the application grow significantly larger over time. Typically, the individual things do not get significantly larger; we simply get more and more of them. It really doesn't matter what resource on the computer is saturated first, the increase in demand will drive us to spread what formerly ran on a small set of machines to run over a larger set of machines… [2]

  Almost-infinite scaling is a loose, imprecise, and deliberately amorphous way to motivate the need to be very clear about when and where we can know something fits on one machine and what to do if we cannot ensure it does fit on one machine. Furthermore, we want to scale almost linearly[3] with the load (both data and computation).

- 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 only manipulate one entity at a time. A consequence of almost-infinite scaling is that this programmatic abstraction must be exposed to the developer of business logic.

  By naming and discussing this as-yet-unnamed concept, it is hoped that we can 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 on the messaging patterns used to connect the entities. These lead to the creation of state machines that cope with the message delivery inconsistencies foisted upon the innocent application developer as they attempt to build scalable solutions to business problems.

Assumptions

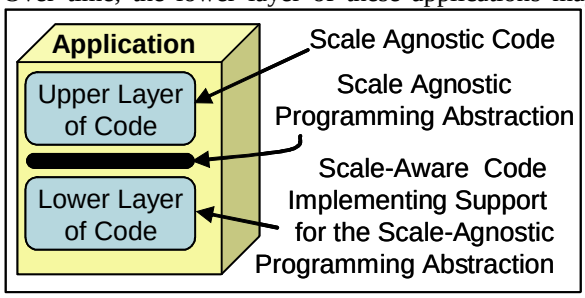
Let's start out with three assumptions which are asserted and not justified. We simply assume these are true based on experience.

- Layers of the Application and Scale-Agnosticism
  Let's start by presuming (at least) two layers in each scalable application. These layers differ in their perception of scaling. They may have other differences but that is not relevant to this discussion.

  The lower layer of the application understands the fact that more computers get added to make the system scale. 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. We are presuming that the lower layer provides a *scale-agnostic programming abstraction* to the upper layer[4].

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

  Over time, the lower layer of these applications may



  evolve to become new platforms or middleware which simplify the creation of scale-agnostic applications (similar to the past scenarios when CICS and other TP-Monitors evolved to simplify the creation of applications for block-mode terminals).

  The focus of this discussion is on the possibilities posed by these nascent scale-agnostic APIs.

- Scopes of Transactional Serializability
  Lots of academic work has been done on the notion of providing transactional serializability across distributed systems. This includes 2PC (two phase commit) which can easily block when nodes are unavailable and other protocols which do not block in the face of node failures such as the Paxos algorithm.

[2] To be clear, this is conceptually assuming tens of thousands or hundreds of thousands of machines. Too many to make them behave like one "big" machine.

[3] Scaling at N log N for some big log would be really nice…

[4] Google's MapReduce is an example of a scale-agnostic programming abstraction.

Let's describe these algorithms as ones which provide *global transactional serializability*[5]. Their goal is to allow arbitrary atomic updates across data spread across a set of machines. These algorithms allow updates to exist in a single scope of serializability across this set of machines.

We are going to consider what happens when you simply don't do this. Real system developers and real systems as we see them deployed today rarely even try to achieve transactional serializability across machines or, if they do, it is within a small number of tightly connected machines functioning as a cluster. Put simply, we aren't doing transactions across machines except *perhaps* in the simple case where there is a tight cluster which looks like one machine.

Instead, we assume *multiple disjoint scopes of transactional serializability*. Consider each computer to be a separate scope of transactional serializability[6].

Each data item resides in a single computer or cluster[7]. Atomic transactions may include any data residing within that single scope of transactional serializability (i.e. within the single computer or cluster). You cannot perform atomic transactions across these disjoint scopes of transactional serializability. That's what makes them disjoint!

- Most Applications Use "At-Least-Once" Messaging
  TCP-IP is great if you are a short-lived Unix-style process. But let's consider the dilemma faced by an application developer whose job is to process a message and modify some data durably represented on disk (either in a SQL database or some other durable store). The message is consumed but 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 in which the message is delivered more than once. The messaging plumbing does this because its only other recourse is to occasionally lose messages ("at-most-once" messaging) and that is even more onerous to deal with[8].

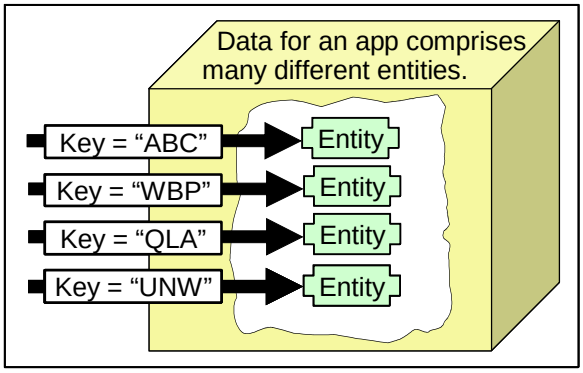
  A consequence of this behavior from the messaging plumbing is that the application must tolerate message retries and the out-of-order arrival of some messages. This paper considers the application patterns arising when business-logic programmers must deal with this burden in almost-infinitely large applications.

Opinions to Be Justified

The nice thing about writing a position paper is that you can express wild opinions. Here are a few that we will be arguing in the corpus of this position paper[9]:

- Scalable Apps Use Uniquely Identified "Entities"
  This paper will argue that the upper layer code for each application must manipulate a single collection of data we are calling an *entity*. There are no restrictions on the size of an entity except that it must live within a single scope of serializability (i.e. one machine or cluster).

  Each entity has a unique identifier or key. An entity-key may be of any shape, form, or flavor but it somehow uniquely identifies exactly one entity and the data contained within that entity.



  There are no constraints on the representation of the entity. It may be stored as SQL records, XML documents, files, data contained within file systems, as blobs, or anything else that is convenient or appropriate for the app's needs. One possible representation is as a collection of SQL records (potentially across many tables) whose primary key begins with the entity-key.

[5] I am deliberately conflating strict serializability and the weaker locking modes. The issue is the scope of the data participating in the transactions visible to the application.

[6] This is not intended to preclude a small collection of computers functioning in a cluster to behave as if they are one machine. This IS intended to formally state that we assume many computers and the likelihood that we must consider work which cannot be atomically committed.

[7] This is excluding replication for high-availability which will not change the presumption of disjoint scopes of tra`nsactional serializability.

[8] I am a big fan of "exactly-once in-order" messaging but to provide it for durable data requires a long-lived programmatic abstraction similar to a TCP connection. The assertion here is that these facilities are rarely available to the programmer building scalable applications. Hence, we are considering cases dealing with "at-least-once".

[9] Note that these topics will be discussed in more detail.

Entities represent disjoint sets of data. Each datum resides in exactly one entity. The data of an entity never overlaps the data of another entity.

An application comprises many entities. For example, an "order-processing" application encapsulates many orders. Each order is identified by a unique Order-ID. To be a scalable "order-processing" application, data for one order must be disjoint from the data for other orders.

- Atomic Transactions Cannot Span Entities
  We will argue below why we conclude that atomic transactions cannot span entities. The programmer must always stick to the data contained inside a single entity for each transaction. This restriction is true for entities within the same application and for entities within different applications.

  From the programmer's perspective, *the uniquely identified entity is the scope of serializability*. This concept has a powerful impact on the behavior of applications designed for scaling. An implication of this we will explore 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 which is then consumed by a stateless process.

  Standard practice is to include some data in the message that informs the stateless application code where to get the data it needs. This is the entity-key described above. The data for the entity is fetched from some database or other durable store by the application.

  A couple of interesting trends are already happening in the industry. First, the size of the set of entities within a single application is growing larger than can fit in a single data-store. Each individual entity fits in a store but the set of them all does not. Increasingly, the stateless application is routing to fetch the entity based upon some partitioning scheme. Second, the fetching and partitioning scheme is being separated into the lower-layers of the application and deliberately isolated from the upper-layers of the application responsible for business logic.

  This is effectively driving towards the message destination being 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 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 (as manifest by its durable state and identified by its entity-key) sends a message which is addressed to another entity. The recipient entity comprises both upper-layer (scale-agnostic) business logic and the durable data representing its state which is stored and accessed by the entity-key.

  Recall the assumption that messages are delivered "at-least-once". This means that the recipient entity must be prepared in its durable state to be assailed with redundant messages that must be ignored. In practice, messages fall into one of two categories: those that affect the state of the recipient entity and those that do not. Messages that do not cause change to the processing entity are easy… They are trivially idempotent. It is those making changes to the recipient that pose design challenges.

  To ensure idempotence (i.e. guarantee 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, the repeated message will typically generate a new response (or outgoing message) which mimics the behavior of the earlier processed message.

  The knowledge of the received messages creates state which is wrapped up on a per-partner basis. The key observation here is that the state gets organized by partner and the partner is an entity.

  We are applying the term activity to the state which manages the per-party messaging on each side of this two-party relationship. Each activity lives is exactly one entity. An entity will have an activity for each partner entity from which it receives messages.

  In addition to managing message melees, activities are used to manage loosely-coupled agreement. 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.

  This paper is not asserting that activities can solve the well known challenges to reaching agreement described so thoroughly in workflow discussions. We are, however, pointing out that 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 what we call an activity.

  Examples of activities are sometimes subtle. An order application will send messages to the shipping application and include 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 the design for activities.

The remaining part of this paper will examine these assertions in greater depth and propose arguments and explanations for these opinions.

2. ENTITIES

This section examines the nature of entities in greater depth. We first consider the guarantee of atomic transactions *within* a single entity. Next, we consider the use of a unique key to access the entity and how this can empower the lower-level (scale-aware) part of the application to relocate entities when repartitioning. After this, we consider what may be accessed within a single atomic transaction and, finally, examine the implications of almost-infinite scaling on alternate indices.

Disjoint Scopes of Serializability

Each entity is defined as a collection of data with a unique key known to live within a single scope of serializability. Because it lives within a single scope of serializability, we are ensured that we may always do atomic transactions *within a single entity*.

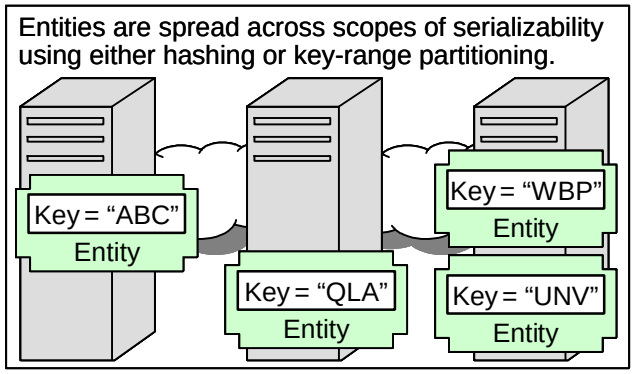
It is this aspect that warrants giving the "entity" a different name then an "object". Objects may or may not share transactional scopes. Entities never share transactional scopes because repartitioning may put them on different machines.

Uniquely Keyed Entities

Code for the upper layer of an application is naturally designed around collections of data with a unique key. We see customer-ids, social-security-numbers, product-SKUs, and other unique identifiers all the time within applications. They are used as keys to locate the data implementing the applications. This is a natural paradigm. We observe that the boundary of the disjoint scope of serializability (i.e. the "entity") is always identified by a unique key in practice.

Repartitioning and Entities

One of our assumptions is that the emerging upper-layer is scale-agnostic and the lower-layer decides how the deployment evolves as requirements for scale change. This means that 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.



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 scope of serializability[10]. How can you know that two separate entities are guaranteed to be within the same scope of serializability (and, hence, atomically updateable)? You only know when there is a single unique key that unifies both. Now it is really one entity!

If we use hashing for partitioning by entity-key, there's no telling when two entities with different keys land on the same box. If we use key-range partitioning for the entity-keys, most of the time the adjacent key-values resides on the same machine but once in a while you will get unlucky and your neighbor will be on another machine. A simple test-case which counts on atomicity with a neighbor in a key-range partitioning will very likely experience that atomicity during the test deployment. Only later when redeployment moves the entities across different scopes of serializability will the latent bug emerge as the updates can no longer be 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 each entity-key (and its entity) reside on a single machine (or small cluster). Different entities may be anywhere.

A scale-agnostic programming abstraction must have the notion of entity as the boundary of atomicity. The understanding of the existence of the entity as a programmatic abstraction, the use of the entity-key, and the clear commitment to assuming a lack of atomicity across entities are essential to providing a scale-agnostic upper layer to the application.

Large-scale applications implicitly do this in the industry today; we just don't have 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 serializability. Assuming more will break when the deployment changes.

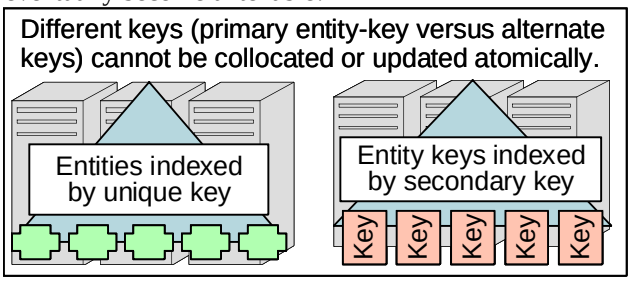
Considering Alternate Indices

We are accustomed to the ability to address data with multiple keys or indices. For example, sometimes we reference a customer by social security number, sometimes by credit-card number, and sometimes by street address. If we assume extreme amounts of scaling, these indices cannot reside on the same machine or a single large cluster. *The data about a single customer*

[10] Recall the premise that almost-infinite scaling causes *the number of entitys* to inexorably increase but *size of the individual entity* remains small enough to fit in one scope of serializability (i.e. one computer or small cluster).

cannot be known to reside within a single scope of serializability! The entity itself can reside within a single scope of serializability. The challenge is that the copies of the information used to create an alternate index must be assumed to reside in a different scope of serializability!

Consider guaranteeing the alternate index resides in the same scope of serializability. When almost-infinite scaling kicks in and the set of entities is smeared across gigantic numbers of machines, the primary index and alternate index information must reside within the same scope of serializability. How do we ensure that? The only way to ensure they both live within the same scope is to begin locating the alternate index using the primary index! That takes us to the same scope of serializability. If we start without the primary index and have to search all of the scopes of serializability, 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, we lookup the alternate key and that yields the entity-key. Second, we 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 our premise of almost-infinite scaling means the two indices (primary and alternate) cannot be known to reside in the same scope of serializability!

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).

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 to the almost-infinite scale application. Reading of the data that was previously kept as alternate indices must now be done with an understanding that this is potentially out of sync with the entity implementing the primary representation of the data. The functionality previously implemented as alternate indices is now harder. It is a fact of life in the big cruel world of huge systems!

3. MESSAGING ACROSS ENTITIES

In this section, we consider the means to connect independent entities using messages. We examine naming, transactions and messages, look at message delivery semantics, and consider the impact of repartioning the location of entities on these message delivery semantics.

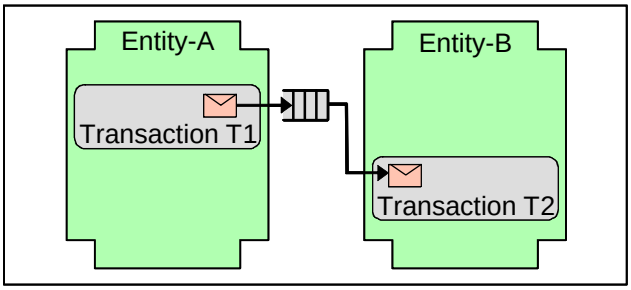
Messages to Communicate across Entities

If you can't update the data across two entities in the same 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 in another entity. By the definition of an entity, we must assume that they cannot be atomically updated.

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



If the message cannot be seen at the destination until after the sending transaction commits, we see the message as 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 into a new entity causing transactions.

Naming the Destination of Messages

Consider the programming of the scale-agnostic part of 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 data

for the entity and, hence, the destination for 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, we see scale-agnostic applications are evolving to support idempotent processing of all application-visible messaging[11]. This implies reordering in message delivery, too.

4. ENTITIES, SOA, AND OBJECTS

This section contrasts the ideas in this paper to those of object orientation and service orientation.

Entities and Object Instances

One may ask: "How is an entity different than an object instance?" The answer is not black and white. Objects have many forms, some of which are entities and others which are not. There are two important clarifications that must be made to consider an object to be an entity.

First, the data encapsulated by the object must be strictly disjoint from all other data. Second, that disjoint data may never be atomically updated with any other data.

Some object systems have ambiguous encapsulation of database data. To the extent these are not crisp and diligently enforced; these objects are not entities as defined herein. Sometimes, materialized views and alternate indices are used. These won't last when your system attempts to scale and your objects aren't entities.

Many object systems allow transaction scopes to span objects. This programmatic convenience obviates most of the challenges described in this paper. Unfortunately, that doesn't work under almost-infinite scaling unless your transactionally-coupled objects are always collocated[12]. To do this, we need to assign them a common key to ensure co-location and then realize the two transactionally-coupled objects are part of the same entity!

Objects are great but they are a different abstraction.

Messages versus Methods

Method calls are typically synchronous with respect to the calling thread. They are also synchronous with respect to the calling object's transaction. While the called object may or may not be atomically coupled with the calling object, the typical method call does not atomically record the intent to invoke a message and guarantee the at-least-once invocation of the called message. Some systems wrap message-sending into a method call and I consider those to be messages, not methods.

We don't address the differences in marshalling and binding that usually separate messaging from methods. We simply point out that transactional boundaries mandate asynchrony not usually found with method calls.

Entities and Service Oriented Architectures

Everything discussed in this paper is supportive of SOA. Most SOA implementations embrace independent transaction scopes across services.

The major enhancement to SOA presented here is the notion that each service may confront almost-infinite scaling within itself and some observations about what that means. These observations apply across services in a SOA and within those individual services where they are designed to independently scale.

5. ACTIVITIES: COPING WITH MESSY MESSAGES

This section discusses means to cope with the challenges of message retries and reordering. We introduce the notion of an activity as the local information needed to manage a relationship with a partner entity.

Retries and Idempotence

Since any message ever sent may be delivered multiple times, we need a discipline in the application 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 due to 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. So far, this is not yet available. Hence, we consider what the poor developer of the scale-agnostic application must implement.

[11] It is common that scale-aware applications are not initially designed for idempotence and re-ordering of messages. At first, small scale deployments do not exhibit these subtle problems and work fine. Only as time passes and their deployments expand do the problems manifest and the applications respond to handle them.

[12] Alternatively, you could forget about collocation and use two phase commit. Per our assumptions, we assert natural selection will kick in eliminating this problem…

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 which 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. We consider this to be true even if a log record describing the read is written. The log record is not substantive to the behaviour of the entity. The definition of what is and what is not substantive is application specific.[13]

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 way that is substantive? Those, too, would be naturally idempotent.

Now, it gets harder. The work implied by some messages actually cause substantive changes. These messages are not naturally idempotent. The application must include mechanisms to ensure that these, too, are idempotent. This means remembering in some fashion that the message has been processed so that subsequent attempts make no substantive change.[14]

It is the processing of messages that are not naturally idempotent that we consider next.

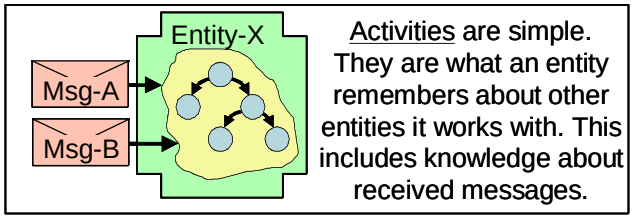
Remembering Messages as State

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 to remembering that a message has been processed, if a reply is required, the same reply must be returned. After all, we don't know if the original sender has received the reply or not.

Activities: 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*. Each entity may have many activities if it interacts with many other entities. Activities track the interactions with each partner.

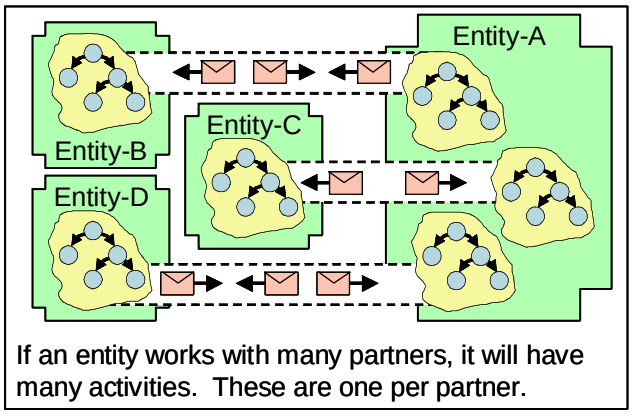


Each entity comprises a set of activities and, perhaps, some other data that spans the activities.

Consider the processing of an order comprising 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 because 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 with the entity-keys. Because the partner is a long ways away, you have to formally manage your understanding of the partners state as new knowledge of the partner arrives. The local information that you know about a distant partner is referred to as an activity.



[13] In the database community, we frequently use this technique. For example, in a physiological logging (ARIES-style) system, a logical undo of a transaction will leave the system with the same records as before the transaction. In doing so, the layout of the pages in the Btree may be different. This is not substantive to the record-level interpretation of the contents of the Btree.

[14] Note that it is hard (but certainly doable) to build the plumbing to eliminate duplicates as messages flow between fine-grained entities. Most durable duplicate elimination is done on queues with a coarser granularity than the entity. Migrating the state for duplicate elimination with the entity isn't commonly available. Hence, a common pattern is that the scale-agnostic application contains application specific logic to ensure redundant processing of messages has no substantive impact on the entity.

Ensuring At-Most-Once Acceptance via Activities

Processing messages that are not naturally idempotent requires ensuring each message is processed at-most-once (i.e. the substantive impact of the message must happen at-most-once). To do this, there must be some unique characteristic of the message that is remembered to 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 will use 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. By leveraging a per-partner collection of state, the programmer can focus on the per-partner relationship.

The assertion is that by focusing in on the per-partner information, it is easier to build scalable applications. One example is in the implementation of support for idempotent message processing.

6. ACTIVITIES: COPING WITHOUT ATOMICITY

This section addresses how wildly scalable system make decisions without distributed transactions.

The emphasis of this section is that it is hard work to manage distributed agreement. In addition, though, 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 we must accept uncertainty as we attempt to come to decisions across different entities. It is unavoidable that decisions across distributed systems involve accepting uncertainty for a while[15]. When distributed transactions can be used, that uncertainty is manifest in the locks held on data and is managed by the transaction manager.

In a system which 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. Nothing magic, just that we can't use distributed transaction so we need to use workflow.

The assumptions that lead us to entities and messages, lead us to the conclusion that the scale-agnostic application must manage uncertainty itself using workflow if it needs 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. The semantics of uncertainty is wrapped up in the behaviour of the business functionality. While more complicated to implement than simply 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 one can visualize that 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. As each partner advances into a new state, the activity tracks this.

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 on, 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. Within each item will be information about outstanding orders against that item. Each of these activities within the item (one per order) manages the uncertainty of the order.

Performing Tentative Business Operations

To reach an agreement across entities, one entity has to ask another to accept some uncertainty. This is done by sending a message which requests a commitment but leaves open the possibility of cancellation. This is called a *tentative operation* and it represented by a message flowing between two entities. At the end of this step, one of the entities agrees to abide by the wishes of the other[16].

Tentative Operations, Confirmation, and Cancellation

Essential to a tentative operation, is the right to cancel. Sometimes, the entity that requested the tentative operation decides it is not going to proceed forward. That is a *cancelling operation*. When the right to cancel is released, that is a *confirming operation*. Every tentative operation eventually confirms or cancels.

When an entity agrees to perform a tentative operation, it agrees to let another entity decide the outcome. This is accepting uncertainty and adds to the general confusion experience by that entity. As confirmations and cancellations arrive, that decreases

[15] At the time of writing this paper, funny action at a distance has not been proven and we are limited by the speed of light. There ain't no such thing as simultaneity at a distance…

[16] This is the simple case. In some cases, the operations can be partially handled. In other cases, time-outs and/or reneging can cause even more problems. Unfortunately, the real world is not pretty.

uncertainty. It is normal to proceed through life with ever increasing and decreasing uncertainty as old problems get resolved and new ones arrive at your lap.

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

Uncertainty and Almost-Infinite Scaling

The interesting aspect of this for scaling is the observation that the management of uncertainty usually revolves around two-party agreements. It is frequently the case that multiple two-party agreements happen. Still, 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. So does the seller, the mortgage company, and all the 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 you consider 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) we see the basis for how distributed agreement is achieved. Just as in the escrow company, 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 data lives and must assume it is far away, it can be programmed in a scale-agnostic way.

Real world almost-infinite scale applications would love the luxury of a global scope of serializability as is promised by two phase commit and other related algorithms. Unfortunately, the fragility of these leads to unacceptable pressure on availability. Instead, the management of the uncertainty of the tentative work is foisted clearly 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.

7. CONCLUSIONS

As usual, the computer industry is in flux. One emerging trend is for an application to scale to sizes that do not fit onto a single machine or tightly-coupled set of machines. As we have always seen, specific solutions for a single application emerge first and then general patterns are observed. Based upon these general patterns, new facilities are built empowering easier construction of business logic.

In the 1970s, many large-scale applications struggled with the difficulties of handling the multiplexing of multiple online terminals while providing business solutions. Emerging patterns of terminal control were captured and some high-end applications evolved into TP-monitors. Eventually, these patterns were repeated in the creation of developed-from-scratch TP-monitors. These platforms allowed the business-logic developers to focus on what they do best: develop business logic.

Today, we see new design pressures foisted onto programmers that 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.

Unfortunately, programmers striving to solve business goals like eCommerce, supply-chain-management, financial, and health-care applications increasingly need to think about scaling without distributed transactions. They do this because attempts to use distributed transactions are too fragile and perform poorly.

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 paper argues that these nascent patterns can be applied more consistently in the hand-crafted development of applications designed for almost-infinite scaling. Furthermore, in a few years we are likely to see the development of new middleware or platforms which provide automated management of these applications and eliminate the scaling challenges for applications developed within a stylized programming paradigm. This is strongly parallel to the emergence of TP-monitors in the 1970s.

In this paper, we have introduced and named a couple of formalisms emerging in large-scale applications:

- Entities are collections of named (keyed) data which may be atomically updated within the entity but never atomically updated across entities.
- Activities comprise the collection of state within the entities used to manage messaging relationships with a single partner entity.

Workflow to reach decisions, as have been discussed for many years, functions within activities within entities. It is the fine-grained nature of workflow that is surprising as one looks at almost-infinite scaling.

It is argued that many applications are implicitly designing with both entities and activities today. They are simply not formalized nor are they consistently used. 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.