Autonomous Computing

It is common to compute across autonomous boundaries. The patterns we use to cope with autonomy are rarely discussed even though they frequently impact how we use data and do work.

By Pat Helland – March, 2022

Abstract

Over 23 years ago in 1999, I started speaking about how autonomy impacts our designs. By 2001, I presented these ideas at conferences including HPTS 2001. I regret not setting them to prose. Many of these ideas moved forward into what was called SOA or Service Oriented Architectures in the early 2000s. Now, in 2022, most of these principles still influence our designs but, in my opinion, we don't talk about them or their implications enough.

In this paper, I introduce a concept called a **fiefdom**. This is a collection of autonomous computing and data that fundamentally assumes anyone working with the fiefdom is an outsider. We are autonomous from them. This is not the same as Byzantine consensus since, well, we don't really draw consensus across these boundaries.

Another related computational component is an **emissary**. While an emissary is designed and tested as a part of the fiefdom, it is assumed to be an outsider since it runs outside the autonomous fiefdom. Instead, the emissary knows how to interact with the fiefdom to make the work easier. The emissary, running in close proximity to the fiefdom's partners provides an easy-to-use interface for partners. Because the messages generated by the emissary match the expected patterns for the fiefdom, more of the incoming work meets the fiefdom's expectations and more successful business is transacted. There can even be an emissary embedded within a foreign fiefdom to help the embedding fiefdom work with the emissary's native fiefdom.

Working across boundaries frequently happens over time. This necessitates multiple message interactions that are correlated to each other. Before computers, this was done with paper forms containing unique serial numbers, multiple parts to the form, and successive modifications over time. We will introduce an abstraction called a **collaboration** to explain how this works in many environments. Like fiefdoms and emissaries, this is a pattern with many possible implementations.

These patterns are rooted in centuries of business practices as companies worked across autonomous boundaries. While all the benefits I discussed 20 years ago still apply, there are even more ways to leverage these patterns in our distributed and replicated world. This paper is finally the prose version of my many presentations on Autonomous computing. In addition, I'm adding more thoughts how the patterns described here influence our internal management of enterprise data and how these patterns can help with distribution and scale.

TABLE OF CONTENTS

1. Introduction

Autonomous computing starts with independent computer systems, whimsically named **fiefdoms**. Fiefdoms are independently controlled and managed. They are independent of outsiders. This is not really about identity, authentication, authorization, or secure communications. Those are incredibly important topics but not what this paper addresses. This is more about the flow of messages across autonomous boundaries, the data contained in those messages, and how to successful accomplish business while remaining independent. In this design pattern, the set of related messages used to do a single related business operation is called a **collaboration**. Another part of the pattern lies in **emissaries**, another whimsically named role for code running outside of the fiefdom's autonomous boundary but with clear understanding of what interactions are likely to succeed when working with the fiefdom.

We are going to examine how such autonomous systems interact, how data is handled, and how we can get work done across boundaries. These concepts are essential to distributed work, long-running work, scalable work, and even replicated work and reconciliation.

When originally discussed over 20 years ago, fiefdoms were conceived of as a single database surrounded and protected by a two-tier application. As time has moved on, I see examples of fiefdoms literally running over hundreds of thousands of computers implemented internally to provide massive scale. We will discuss this more later in the paper but fundamentally, the autonomous computing pattern is agnostic to scale both large and small. What it does imply, though, is the importance of working across multiple interactions and correlating later messages to earlier ones using the abstraction we're calling a collaboration. Collaborations encompass the business and domain specific sequences of messages, how they interact, and how they are resolved. Finally, how can the protocols between fiefdoms and their emissaries support scale as well as offline and replicated emissaries?

Within this paper, I will introduce *fiefdoms, collaborations, and emissaries* and show examples within computing and within our daily lives. I examine more deeply how *emissaries* work outside the autonomous boundary and convenient even though they are independent. This paper examines how work *across different fiefdoms* can be initiated, run for long periods of time, and eventually work to completion.

Next, we examine how work happens *within a fiefdom* with **activity code and data** dedicated to work interacting with partners and **resource code and data** focusing in on the artifacts managed by a fiefdom.

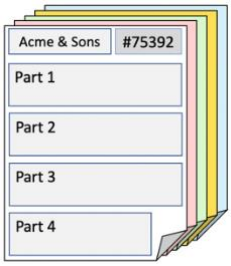
Following this, we disassemble what it means to have **scalable solutions** and how these patterns can empower scale. We look at how collaborations, if done correctly, support **offline work and reconciliation** by ensuring work moves forward and doesn't need to retract any commitments.

Finally, we conclude by reiterating that autonomy and shared work are based on both **collaborations** and the **data shared** between partners.

2. Fiefdoms, Collaborations, and Emissaries

Let's introduce a few concepts seen in the common pattern I'm calling **Autonomous Computing**.

Fiefdom is an English word for am estate with its own independent leadership and rules. It also means a person or entity's sphere of control. Historically, this included duchies, baronies, or other independent legal domains. Typically, a fiefdom has its own customs, laws, and way of doing things. It is largely independent of other places.



Collaborations support messaging over time. Similar to paper forms that empowered business before computers, a collaboration is modified by both emissaries and fiefdoms as they send messages back and forth. In addition to providing the association across these related messages, they provide a means to connect the work within a fiefdom or emissary to the long-running internal state that supports the cooperative business work.

An **emissary** is a representative. It is used to describe a person sent on a special mission as a diplomatic representative. Unlike an ambassador, an emissary has NO authority. They can only play nicely, suck-up, and try to ease the relationship.

I am describing **computing patterns** I see in many existing applications. These patterns may be supported by relational data, No SQL key-values, and many other techniques. What I believe is happening is that the combination of related messages and the work they stimulate empowers cooperative business work *even when the partners are independent and autonomous*.

Let's first introduce the notion of a fiefdom within the **autonomous computing pattern**. After this, we'll consider how a special kind of software called an emissary, designed with the fiefdom in mind, can ease the challenges of working with the emissary's fiefdom. The collaboration messaging pattern gets our attention next. After that, we sketch the basic framework for offline processing. Finally, we introduce how the rigid and isolating boundary providing autonomy to a fiefdom changes the nature of data both inside and outside of that fiefdom.

2.1. Fiefdoms and Autonomous Computing

The only way to get work done with a fiefdom is by sending a message with great humbleness and supplication. The fiefdom describes the messages it is willing to process and what is required to get work done in cooperation with the fiefdom.

I don't know what it's like for you, but when I go to my bank's ATM machine to withdraw cash, there are only a few things the darned box lets me do. I can withdraw cash, deposit a check, transfer funds, and check my balance. All attempts to get a JDBC connection to the bank's

backend database have failed! It's like my bank views me as an outsider! How do we work together when one party doesn't trust the other? What about when neither party trusts the other?

This is what it's like working with a fiefdom. A fiefdom can be implemented with computing software and hardware or it can be a bunch of people trying to do business together without computers.

Requesting Service from a Fiefdom

Fiefdom's typically define their own rules for how to get work done. If you want cash, put in your ATM card and your PIN. If you want to sell your products on the shelf of my mega-store, send me the following messages with electronic descriptions of the products and begging my permission. In general, the shape and description of the interaction is defined by the fiefdom.

As we will see below, sometimes multiple fiefdoms work together. On a whole, the fiefdom with the most economic sway in the relationship defines the protocol and messages to be used in the interaction. If you want a ginormous bricks and mortar retailer to sell your product, you probably need to interact with them by THEIR RULES. If you're a humongous manufacturer, a tiny retailer will send you whatever messages YOU desire. The economic dog wags the economic tail.

Private Data within Fiefdoms

Encapsulated within the fiefdom is private data. This is internal and, in general, is not shared with outsiders. My bank knows my accounts and their balances. It knows if I've overdrawn my accounts or can manage my money and do arithmetic. Based on this information, it may allow larger withdrawals or even give me money if the ATM is disconnected from the bank's datacenter. It will NOT tell ME this information about YOUR accounts. Heck, it's unlikely to tell ME about MY bank accounts other than the minimal information possible.

This private data encapsulated within the fiefdom may be varied and widespread. It contains the fiefdom's most precious thoughts and business knowledge. This is internal and should remain internal stuff.

Transactions and Fiefdoms

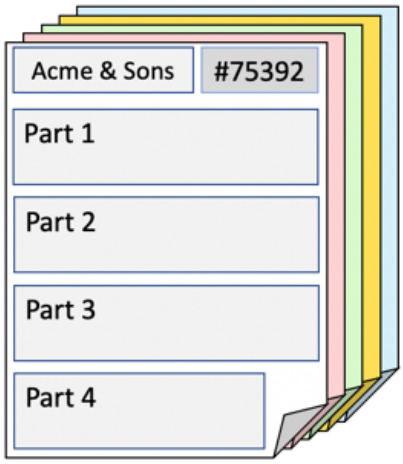
Transactions across boundaries create dependencies across boundaries. Classic database transactions require some form of coordination and usually cause data records to be locked up awaiting the transaction's outcome. As an autonomous fiefdom, no way in heck am I going to lock up MY database records awaiting YOU while you take your sweet time! That just means I can't do my work while you take a vacation! That ain't gonna happen!

Because we are independent, we have no distributed transactions. Each fiefdom may use database transactions INSIDE its belly. There is no way is it going to share a transaction with some miscreant business partner knocking at the gate of the castle. Nope… No way!

Fiefdoms are a design pattern for long-running interactions *across autonomous boundaries*. Work happens with a sequence of related messages over time that ensures cooperative work is accomplished. This is how it was done centuries ago and it's how it's done today.

2.2. Collaborations for Long-Running Work

A **collaboration** is an abstraction for a set of messages into and out of a fiefdom for a single long-running business operation. This is not a new idea. It simply got forgotten by computer scientists and software engineers as they focused in on the problems with a SINGLE COMPUTER and then later by the work in distributed systems which tacitly tried to take the single computer and spread its work around many machines.



Back when I was a kid in the 1960s, there were no computers in our daily lives. Sure, large companies had them in the back rooms and the government and military had big mainframes. Still, we had no interaction with computers as we moved through our lives. Instead, we had paper. When shopping, going to the doctor's office, or getting something repaired, you filled out forms that controlled the activity. I vividly remember the multi-part and multi-colored forms required to get something done.

Sometimes, I would accompany one of my parents on some errand, dropping off dry cleaning, getting the car repaired, or ordering something at the department store. In all cases, a specialized form was pulled from below the counter and my mother or father would fill out Part-1 on the front page being careful to push really hard with the pen to ensure the writing pushed back to the last page. Each of these forms came pre-printed with a serial number in the upper right corner. When my parent finished filling out Part-1, the back page of the multi-colored form was torn off and handed back to them. The rest of the pages were kept at the store or repair shop. They would do whatever was required and fill in the other parts of the form. Different pages (with their unique colors) were kept within the departments to ensure every participant had a record. These forms knit together the work of the internal departments of the business and tied back to the customer (typically my mother).

The second from the back page was for the front desk. It was torn off and filed by the serial number into a file cabinet. The rest of the form went into the outbox on the desktop to be routed to the business's internal department who would do the next part of the work, fill out Part-2, tear off the back page of the form, and send the work off on its journey. The front desk's file folder was organized by serial number (imprinted on the top right of the form) with the active work for the business. Only when the work was completed and my mom picked up the purchase, the dry cleaning, or the repaired item did the paper form move into the "completed work" file cabinets, still filed by its serial number.

This multi-part paper form is a **collaboration**. It captures the request, tracks work as it proceeds, and ensures everything it done before the completed paper form is retired into the "completed work" file folder.

Doing work across autonomous boundaries may require multiple interactions. Correlating these interactions used to require paper forms. In computing interactions, multiple messages must also be correlated and tied to earlier or later work. *This is the collaboration pattern.*

Collaborations are implemented within most applications by tracking a unique number within each message. This is used to find the related messages in the collaboration and move the long-running work forward. This is how the abstraction becomes reality today.

Using Reference Data to Fill Out Fields in a Collaboration

One of the highlights of my mom's year was the twice annual Sears catalog. This was a massive book with many hundreds of pages of stuff available for sale. Dresses, kitchen items, household tools, professional tools, you name it and it was in the catalog. Heck, from 1908 to 1940, you could even buy a kit to assemble a new house including instructions for the pouring the foundation! That was before my mom's time as a Sears customer, though.

When the catalog arrived, many of us in the household would study the wonderous array of possible purchases. Included was a form that needed filling out with item-numbers and prices. These would be diligently copied into the form to enumerate the desired stuff. Mom would enclose the form in an envelope and send it to Sears along with a check for payment. Credit cards were not so common at the time. Weeks later, it was like Christmas when a delivery truck brought you all these things that were not easily attainable in your home town.

The information in the Sears catalog was, in my nomenclature, **reference data**. Its sole purpose was to facilitate filling out the order form. Since it only came twice a year, it was somewhat stale with passing time. The catalog clearly stated the timeframe during which the item-numbers and prices could be used for purchasing stuff.

So, I would call this using *reference data* to fill out a *collaboration* to get work done across distrusting boundaries. In this example, Sears mail-order was a fiefdom. We had no idea what internal departments within Sears were involved in fulfilling the order.

Finally, both the reference data and the collaboration had a fixed lifetime. They were created, spread across the trust boundaries, used for a while, and then retired. Eventually, an expired Sears catalog might become a doorstop or get fed into the fireplace.

2.3. Emissaries: Helping the Interaction with Fiefdoms

I remember when I bought my first home in 1979. It was a tiny crappy place that was way too small for a family of six. To get the mortgage, we went to a mortgage broker. They are paid to be nice and make the business easier to understand for the borrower.

In our case, the mortgage broker was NOT employed by any bank. They helped us get a loan from one of many banks based on who would get us the best deal. The mortgage broker had information about various banks including rate sheets, the needed qualifications, and lots of stuff barely understandable to a 23-year-old first time home buyer with a big family.

When a bank was selected, we filled out lots of forms using the reference data cached in the mortgage broker's desk. When this stuff was presented to the bank, they did NOT trust it. A credit check was ordered by the bank. So was the employment check. Everything in the forms

was verified by the distrusting bank. The bank found this convenient since the mortgage broker knew how to fill out the forms correctly making bank's job easier.

A mortgage broker is a real-life **emissary**. They are not within the autonomous boundary of the bank but do a heck of a lot of useful work, possibly working with many different banks.

Within computing systems, we see many uses of the **emissary design pattern**. Shopping at large eCommerce sites is done using an emissary as the front-end. In this case, the emissary is implemented with thousands or tens of thousands of servers supporting shopping carts, product catalogs, images of products, recommendations, reviews, "people who bought this frequently bought that" and much more. In my mind, everything before the shopper pushes "submit" is an emissary pattern. After you push submit, the request is sent to the backend system for payment processing, inventory check, shipment scheduling, and more.

Complex and distributed back-end processing implements the eCommerce fiefdom. Indeed, there will likely be many internal fiefdoms inside the big eCommerce back-end. These follow the autonomous boundaries of departments within the big eCommerce company.

Sometimes Many Emissaries; Sometimes Many Fiefdoms

Consider your mobile phone and its applications. Email comprises both a back-end fiefdom implemented by your mail-server(s) and the emissary on your mobile phone. I have my phone's mail connected to my work's email and two personal accounts. You may use a browser-based mail client that interacts with the same mail servers or a different overlapping set. The protocol between each of the emissaries and the backend fiefdoms is a fancy collaboration for each of the emissary-to-fiefdom pairings. This may seem like a shared database with optimistic concurrency but in reality, it is a protocol in which information is constantly added, including possible tombstone denoting the intention to delete a mail message. Mail messages are never really deleted. If you doubt that, ask any FBI agent. Instead, they are archived with the tombstone indicating they're not to be shown.

Emissaries are NOT trusted by any fiefdom. They are autonomous and everything they do is verified once the request is behind the castle wall and inside the independent computing environment. They may provide the convenient front-end for many different fiefdoms unifying how the user sees them behind a single mortgage broker or a single mail client front-end.

Emissaries and Their Data

Emissaries may have reference data, perhaps in bulk form like the Sears catalog or perhaps in offline images of stale email messages. Also, an emissary may have significant **per-user state** capturing your abilities as a borrower, your shopping cart containing proposed purchases, or your view of the downloaded email messages on your phone.

Emissaries are a design pattern that makes working with fiefdoms easier. They are independent, not trusted by the fiefdom, execute outside the fiefdom they represent, and specialize in simplifying the multi-message collaboration with the fiefdom across its autonomous boundary.

The big key to the pattern comes from how data is used within the multi-message collaboration between the emissary and the fiefdom it is designed to support.

2.4. What Is Offline Processing?

On one hand, offline processing is pretty easy. Just buffer up enough reference data and take it offline, fill out new information into the shared collaboration as changes happen, and squirt it over to the fiefdom when you reconnect. Sitting at my mother's kitchen table filling out forms for the Sears catalog is offline processing. Working with the mortgage broker at their office is offline processing. So is reading and writing email using my mobile phone. Of course, depending on the application, there are additional challenges some of which we will talk about below.

Let's address some of the easier problems first. How can you know what data to bring along? Of course, that's both application (or domain) specific and may be strongly dependent on the user's preferences or their past history using the application.

Recurrent Reference Data

Sometimes, there's a notion of **recurrent reference data** which is something you sign up for on a recurring basis. My mom's biannual Sears catalog was offline recurrent reference data.

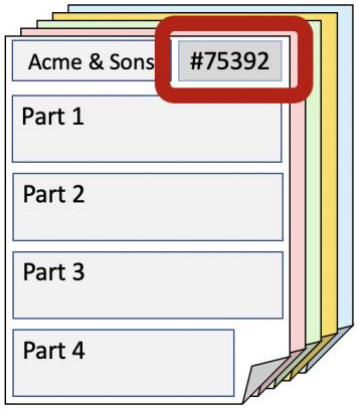
Email is another interesting case of offline recurrent reference data. Most of us may have multiple email accounts held with different mail providers. In each case, an email account has a notion of the set of messages in each account. Using a browser to access my email is typically done for only one mail provider and synchronously fetches a set of messages. Using a mail client on my laptop, tablet, and/or mobile phone caches the offline state of the set of messages. My mail clients all have a notion of the recurrent reference data describing my various email accounts. Each emissary (my device's mail client) collaborates with its fiefdoms (one of the mail servers) whenever it can. The emissary receives updates to its recurrent reference data (the changes to my mail account). Finally, the intermittently offline emissary (the mail client) adds information about its desired work to the collaboration shared by THIS mail client and a specific mail server.

This pattern of recurrent reference data is increasing as the complexity of our work increases. We see many different shared folders, channels, groups, and team-based-blather-sessions used by many people and their devices to facilitate teamwork. Application-to-application usage of recurrent reference data is nascent and emerging.

Using Collaborations for Offline Work

Similar to a paper form, a collaboration provides communication across autonomous boundaries.

Collaborations may be implemented with some buffering within the emissary. Unfortunately, this is typically left as work for the application developer to build since this the autonomous computing pattern is usually hand-crafted within each application. Allowing asynchronous delivery of the messages from the emissary to the fiefdom supports one piece of the puzzle needed for offline work. It is unfortunate that so many of these mechanisms must be re-implemented over and over for different applications.



The abstraction of a collaboration provides ordered full-duplex messaging *within a single collaboration*. Across separate collaborations, all bets are off and no guarantees are made. Within the collaboration, there may be ongoing work for minutes, hours, days, or months. This implies some form of durable storage on each end, both within the fiefdom and within the emissary. There may be many collaborations ongoing at the same time. The important notion is that the domain-specific application defines *its notion of a single ongoing piece of work* and leverages this ordered full-duplex set of messages to get the job done.

When you can't do distributed transactions, you need to manage related messages.

Collaborations simply group related messages for a single job over time. How you do this within the pattern will vary with different implementations.

Richer partnerships for long-running work require richer message patterns. Negotiating for the delivery of thousands of parts in different shipments scheduled to span months is complex. It may need many messages before any shipment happens, many messages during the timeframe of delivery, as well as many messages to resolve the payment schedule and the closure of the work. These are typically correlated with an identifier strongly analogous to the serial number in the upper right-hand corner of the paper forms from the olden-days.

Within the collaboration and its many messages, we may see messages from the participating fiefdoms or emissaries write into different "Parts" of the collaboration. These are simply classes of messages sent by a collaborator for a specific purpose. These may flow in a logically full-duplex fashion with each collaborator loosely-coupled from the other.

It really doesn't matter if a particular implementation pushes or pulls the message across the wire for delivery. What matters is the application semantics that evolve the workflow forward to get the business done!

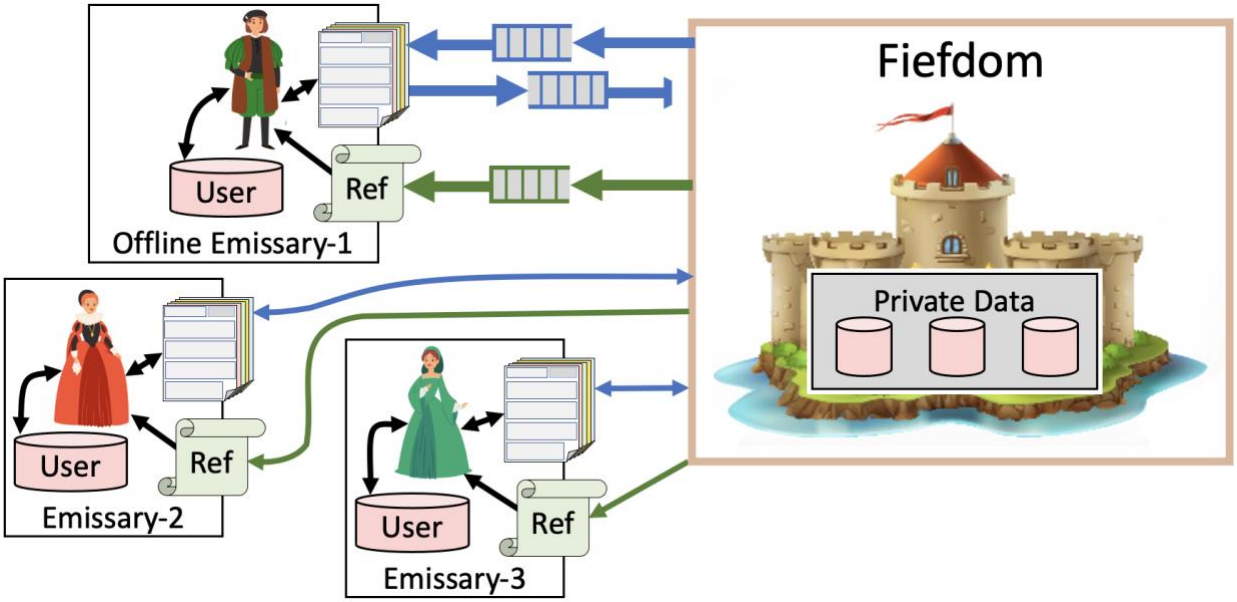
All collaboration-based work is offline! When collaborators (fiefdoms and/or emissaries) don't share transactions, they are ALWAYS offline. Semantically, the work is not atomic and it must ratchet its way forward step-by-step to complete the business task.

2.5. Rethinking Data across Trust Boundaries

So, data has certain roles in this design pattern:

- **Private Data** Inside the Fiefdom
- **Single-User** Emissary Data
- **Reference Data**: Stale Immutable Data
- **Collaborations** for interaction between fiefdoms and emissaries

Offline processing is just asynchronous and buffered changes to a collaboration shared with the fiefdom. Patterns for reconciling these collaborative changes will be discussed below.



There are special roles for data used in the nether world between these collaborating fiefdoms. Like all data flowing outside an autonomous boundary, this data is identified, possibly versioned, immutable (with the versions), and has descriptive metadata bound to it as it is created. This is using data as I described in my 2005 paper Data on the Outside versus Data on the Inside.

In each case, data outside the fiefdom will use identifiers to knit it together:

- **Collaborations** have their messages associated by an identifier, perhaps an order-id.
- **Reference data** will likely have products, prices, and options tied by one or more identifiers. Email knits together with internal unique message-ids. In most cases, this is versioned over time.
- **Single user data** will likely correlate the ongoing business work to both the active collaboration and the reference data. Shopping carts are single user data.

Inside the fiefdom, identifiers tie the collaborations and the ongoing work with internal state.

Both inside and outside the fiefdom, these identifiers may be kept in a relational database or may be kept in some other durable storage. Identities are durable to ensure they are ready for the next interaction with a related collaboration. See my 2019 paper: Identity by Any Other Name.

3. Working ACROSS Fiefdom Boundaries

Let's look more deeply at work across autonomous fiefdom boundaries. We examine the patterns used by collaborations to cooperatively complete work while bounding dependencies on our partners. We accept uncertainty in our plans and engage in an ongoing dance of accepting uncertainty and risk and resolving that uncertainty and risk. Each of these ongoing collaborations has a lifetime that gets resolved, completed, sealed, retired, archived, and eventually deleted.

Finally, we examine how an emissary, designed to support Fiefdom-A may be embedded within Fiefdom-B to help Fiefdom-B's work interacting with Fiefdom-A succeed.

3.1. Working without Shared Transactions

It is a tenet of autonomy you can't count on distributed transactions across boundaries. I am an enthusiastic supporter of using a local transaction to process each message sent into a collaboration or consumed from a collaboration. That's a local transaction, not a distributed or shared transaction. *If you are autonomous, you don't have shared transactions!*

The only alternative to a shared transaction is a shared sequence of messages. That is the **collaboration** we are defining in this paper. The bi-directional sequence of messages is correlated and understood by both collaborators to be messages related to each other over time.

Now, of course, this implies some kind of identity for each of the computing collaborators. This identity must be long-lived so each side can continue the work with the other. In real-life business interactions, there's also some mechanism for ensuring payments can imposed on the partner collaborator. For large business's working together, this is typically a contract with a threat of legal enforcement. For B2C (business-to-consumer) work, the credit card provides the guarantees to allow charges by the business to the consumer's credit card. The consumer may challenge any wayward billing by the business.

Each interaction in the collaboration starts by digging deeper into the solution and completes by digging your way out. When shopping at a large eCommerce site, you browse, check reviews, look at similar products, and add stuff to your shopping cart. These are done using the retailer's emissary that enhances the shopping experience. It's well established that the quality of this offline (or pseudo-offline) experience contributes a LOT to the retailer's profitability.

The emissary for eCommerce shopping acts as if it were offline, because it's not sharing a transaction. You may see: "USUALLY SHIPS IN 24 HOURS". The label tells you *nothing at all* but is incredibly useful. It is a probabilistic statement with no transactional guarantees. It can't offer transactional guarantees because it is not sharing a transaction. Still, I'm always adding or removing items from the cart based on this phrase as I shop for the grandkids. It's useful!

When the shopping cart is full and the user pushes "SUBMIT", lots of steps begin. Usually, the backend work is asynchronous over seconds, minutes, or days. Here are some typical steps:

- **Emissary prepares to start a collaboration:** It wraps up the shopping cart, user information (including addresses and credit cards) product-ids, offer-ids (i.e., the merchants actually selling the product with their pricing/delivery and promises), gift wrapping requests, and more into a single message.
- **SUBMIT by sending the first message over the collaboration:** This message is enqueued for the order-processing in the back-end at the eCommerce retailer. It is common for the front-end shopping experience to use caching, replication, and other tricks to ensure low-latency communication even at the expense of a "perfect" strongly consistent set of data in the order placed to the back-end. This is discussed more deeply in my paper Mind Your State for Your State of Mind.
- **Async but frequently fast:** Most of the time, you hear a "Bleep" as an email comes to the human user within a few seconds after the SUBMIT. Sometimes, this "Bleep" happens 30 minutes or so later as the back-end order-processing system is busy and the queue backs up.
- **The collaboration begins:** This starts a collaboration to implement this order. The order-ID is then used to correlate all work with this ongoing request. Payment is extracted from the credit card. Messages, sometimes via email, advise the consumer of shipment, delays in shipment, out-of-stock problems, and completion of the order. In sophisticated emissaries, all the collaboration's messages invoke a smart client. Others use email.
- **The collaboration completes:** When the stuff is delivered and the credit card charged, that usually completes the collaboration. In reality, it is kept alive for a while in case the customer cancels the order. Only after 30 or 90 days will the collaboration be complete.

In this example, email messages to the human fulfill some of the collaboration's role. The emissary's reification is a bit fuzzy. Still, the pattern is identical to many business operations.

Collaborations involve tentative work, their possible cancellation, and their hopeful completion over time. Orders may be placed and then cancelled if the retailer runs out of inventory. Shipment dates may be renegotiated. Purchasers may change their mind and return the item for a refund. Credit card charges may be declined.

Cooperative work by distrusting parties involves multiple steps with a framework for incurring financial damage to the other party if necessary. *Correlating related messages is essential.*

The use of an *order-id* or some other identifier to knit together the collaboration is an example of the general concept of using identifiers to connect work in distributed environments.

See Identity by Any Other Name.

3.2. Collaborations, Uncertainty, and Reconciliation

It is extremely common for autonomous fiefdoms to accept and resolve business commitments. The nature of these inherently revolves around the domain of the business, their protocols, and their appetite for risk. Some enterprises will go out on the limb to get business, accepting occasional loss as a part of their life. Others, only want strong assurances to move forward.

In any case, sequences of correlated messages flow between the prospective collaborators.

> It is the nature of these correlated messages, their identity as a set of messages, and the patterns of their use that shape the autonomous computing pattern.

Using a sequence of messages, the communicating fiefdoms (or autonomous boundaries) establish some form of limited trust, usually by ensuring financial damages should it go wrong. Time passes and messages flow to deepen the relationship, accomplish the goals, and back their way out of this single collaboration.

Commitments may fail and the businesses recover. The inventory for your grandkid's birthday present may run dry and you, the grandparent become annoyed. Your credit card may be declined and the retailer is annoyed but has avenues of recourse. The birthday present may ship and be destroyed in transit. The order-ID remains active to use for reconciling the problem. Most likely, the predefined time window for complaints passes and the order-ID is retired.

> It is the correlated sequence of messages we call a **collaboration** that empowers this work across fiefdoms, both B2B (business-to-business) and B2C (business-to-consumer).

Inside the fiefdom, the business may work to manage its risk. This is a balance between the commitments it has made via its collaborations and its challenges meeting those commitments.

Airlines notoriously overbook their seats. This is because passengers notoriously flake out and don't show up. The passengers hope to get a refund for a missed flight so the airlines overbook and pay a penalty if too many passengers DO show up.

Consider hotel reservations. The typical pattern is that a reservation must be confirmed with a credit card and an agreement to be charged unless canceled 72 hours in advance. If the guest does not cancel and does not show up, they usually are changed for one nights stay. Because of the associated fee, hotels do not usually overbook. Rarely, a guest shows up and there's no room due to some problem such as a flood in the room. In general, the hotel will make some special attempt to compensate the guest for this unusual event.

Work across collaborations is really contractual. Either this happens OR THIS PENALTY IS INCURRED. The fiefdom's business plan sets rules and regulations for how it will cooperate over the collaboration and how various contingencies are resolved. This sequence of events happens over time (sometimes months or longer) and time-based exceptions are the norm.

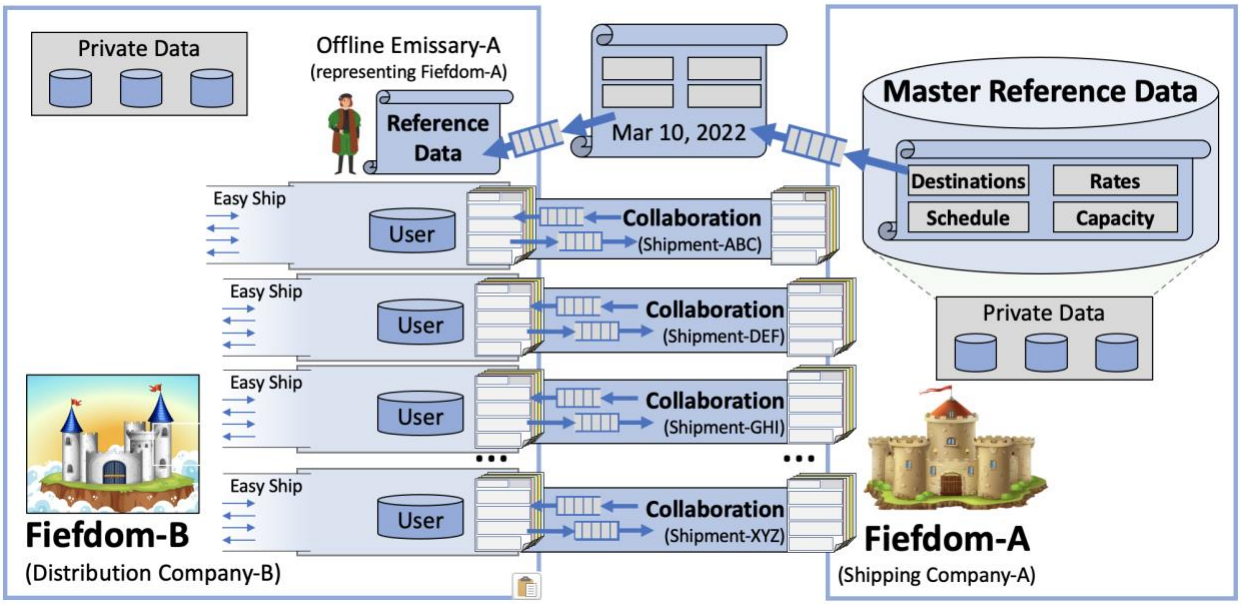
See my 2009 paper Building on Quicksand for more discussion of overbooking, overprovisioning, and other schemes to cope with the uncertainty inherent in long-running autonomous computation.

3.3. Emissaries: Embedding Help in Foreign Places

Sometimes, an emissary is embedded within a foreign fiefdom. This is useful for both fiefdoms as it reduces the challenges when working together. Sometimes, special emissary code for Fiefdom-A runs INSIDE Fiefdom-B to help it reason about collaborations shared with replicated and offline able partners!

Imagine a case where Fiefdom-A (from Company-A) is offering a service to Fiefdom-B (from Company-B). Emissary-A provides helpful code and data caching to navigate the plethora of possible choices available from Fiefdom-A. The code for Emissary-A is inspected as an open source offering and pulled into Fiefdom-B's trust sphere to make life easy as Fiefdom-A and Fiefdom-B do their important work.

For example, a major distribution company (Company-B) has a massive contract with a shipper (Company-A). Company-A's shipping rates, current destinations and schedules, constraints on shipment weights and sizes, and more can be recurrent as reference data constantly published to Company-A's emissary running remotely.



Because Emissary-A can ingest recurrent reference data from its parent Fiefdom-A, there can be sophisticated interactions designed to make it easy to navigate the choices needed in the ongoing business between Company-A.

There may be hundreds of thousands of concurrent collaborations negotiating shipments for thousands of Company-A's customers of its distribution services.

4. Working WITHIN a Fiefdom

So, how do folks design an autonomous fiefdom? How can it support the long-running work necessary to meet its **collaborative commitments** over time?

Typically, the work is broken down into a couple of different roles within the fiefdom.

- **Activities** are data and computation dedicated to tracking the state of the fiefdom's work on behalf of a single collaboration, or a portion of the work for a single collaboration.
- **Resources** are code and data used to manage a shared thing coordinated across multiple activities. These things are frequently tangible physical goods like widgets in inventory or space on a truck needed for a shipment. They may represent more abstract things, too.

These patterns are used to fulfill the external collaboration. They are frequently composed using many internal collaborations. It is VERY common for collaborations, activities, and even resources to have a bounded lifetime. Their lifetime is part of a lifecycle of birth, usage, retirement, archival, and deletion. Indeed, many enterprises and their applications mandate the constrained lifecycle for all their data.

4.1. Inside a Fiefdom: Activities and Activity Data

When an incoming collaboration arrives into the fiefdom, an internal data structure is allocated to track the incoming messages across the collaboration, the work stimulated by these messages, and the completion of that work.

Activities are state machines to track an incoming collaboration and its sequence of messages. Some activities may create new collaborations to talk across other boundaries. Each activity is a long-running state machine supporting the flow of messages across one or more collaborations.

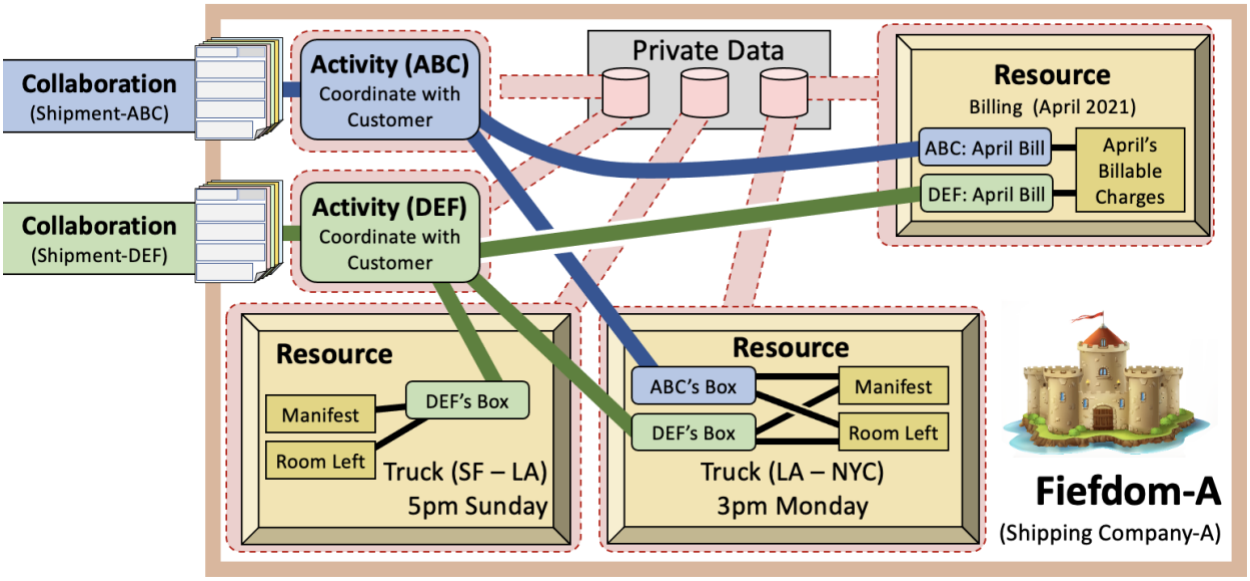
These **activity data structures** are encapsulated by **fiefdom specific activity code**.

Work to advance this activity is stimulated by any of:

- **An external message** arriving over the external collaboration.
- **An internal message** arriving from some internal collaboration used to drive intra-company work.
- **A timer pop** indicating the activity should advance to a new state and perhaps take new actions to manage one of its internal or external collaborations.

Each application fiefdom, in general, hand creates these mechanisms to manage its workflow for activities and the work they spawn. This means an almost infinite variety of ways to make this pattern work. Still, I want to focus on the pattern and its implications.

Activities do not live forever. Their lifecycle is driven by the steps needed to successfully complete business with the partner on the other end of the collaboration. They are created in response to messages within a collaboration, live for a while, perhaps seconds, hours, or months, are retired into a read-only state, archived, and eventually deleted years later.



4.2. Inside a Fiefdom: Resources and Resource Data

Tangible things are managed within **resources**. Inventory, shipments on a truck, passengers booked for a single flight, king-sized-non-smoking-rooms booked for a particular night, billable dollars per customer for a single month. All of these are examples of resources.

It is common that a resource and its surrounding **resource manager** needs to deal with uncertainty. We allocate resources for a customer and its collaboration but need to ensure that the payment arrives first. Perhaps the scheduled space on a truck will not be consumed if the incoming truck is delayed. The resource manager for the outgoing shipment can place a waiting box on the outgoing truck but the activity controlling the waiting box must be notified to adjust the next step on its journey.

Resource managers must work with time, incoming resources, outgoing resources, overprovisioning (to ensure customer happiness) or overbooking (to manage expenses). They cope with an ever-changing world across many competing activities. Sometimes, they will fail to meet obligations to one activity due to the demands of another activity. Sometimes, this is influenced by the importance of the respective customers.

Inside each resource manager are little activities. These track the single allocated resource (or pending resources) that will then be correlated back to a larger scale activity working to advance the work of the collaboration. These, in turn, impact the allocation of the precious resources to other activities.

In general, there may be fine-grained resources under management. For an individual hotel site within a chain, there will likely be resources *for each night in the future for each room class*. King-sized non-smoking ocean-view will be different that double-queen-garden-view. They will each be managed for each separate date that's being scheduled.

It is common for resources and their binding to activities to retire. This depends on both the expiration of the resource (e.g., the flight closed its door and flew away) AND the completion of the activities interacting with it.

Resources live in a constant world of increasing and decreasing uncertainty, especially when overbooking their managed resource. We can only completely resolve the uncertainty of who's on the flight when the plane door shuts and the flight leaves!

One possible design uses collaborations (and their correlated messages) to connect the many activities and resources. This has the advantage of providing unified way to wake up activities, resources, and collaborations in response to an event or a timer popping. As we shall see, the semantic provided by a collaboration may be useful when designing the scalable fiefdom itself!

4.3. The Lifecycle of Data for Autonomous Computing

Everything in this world has a lifecycle of creation, live usage, decline, archival, and retirement. Let's consider the types of data touched by fiefdoms and the lifecycle of data for these types.

The Lifecycle of Reference Data

Reference data is versioned and each version has a bounded lifetime. Fiefdoms generate reference data for the use of incoming collaborations. This reference data is typically versioned and episodically updated and sent out. As mentioned before, there may be a notion of recurrent publication of the reference data so it arrives on a regular basis.

I'm not precluding a fiefdom outsourcing much of the work to generate reference data. Managing product catalogs, price lists, images of products, recommendations, and more is a big task. Caching this information is a large problem. Creating the multiple versions and disseminating the information to the caches is also hard. In addition, interacting with suppliers sending you data that YOU then use to make YOUR reference data is also hard.

When old versions of the reference data expire, they are retired and removed. This may be a fuzzy process when caches and copies are involved. Old versions live side-by-side with new versions as they are shooed out of the system. Users may see jittery views of new and old.

The Lifecycle of a Collaboration

Collaborations have a bounded lifetime. They may be initiated by this fiefdom or they may arrive from outside. Messages flow, work is done, and messages are returned. Sometimes, the work of a collaboration involves nested work with dozens of other companies, all connected via their own separate collaborations.

Eventually, the work of the collaboration winds down. Stuff is delivered and payments are cleared. The allowed time for the returning of stuff or canceling the deal passes. The messaging protocol defined by the collaboration enters a completed state. Everything is done and the collaboration becomes read-only. For a while, its data is rolled up for analysis. Over time, that is less frequent and more course-grain in the analysis.

The collaboration's data structure is retired and archived. It becomes read-only. Eventually, the data in the collaboration is deleted and purged from the system (perhaps years later).

The Lifecycle of Activity Data

Similar to a collaboration, the many activities in the fiefdom are kept alive until the work dies down. When the use of the collaboration that spawned an activity stops, the activity frequently (but not always) goes quiescent, too. Again, activities retire, become read-only, are archived, and eventually the archive is deleted.

The Lifecycle of Resource Data

A bit more surprising is how often resources having a bounded window of life.

For some resources, it makes intuitive sense that their lifetime is bounded. Individual airplane flights and occupancy of a hotel room for a single night become read-only when their date passes.

Surprisingly, it is common to manage long-running resources grouped into time slots.

Your bank account is managed month-by-month. In May, the April bank statement is immutable, read-only, and will never change. If something is wrong, the bank makes a change to May's bank statement. Similarly, inventory in stores is manage by month or year with changes applied as shipments arrive and stuff is sold. Physically counting the stuff on the shelves grounds the truth into time-based realities. New resources within the fiefdom are created after the inventory and the old resource from before the inventory is taken becomes read-only.

It is possible to manage other things in this fashion. Your customer list can and should be managed with periodic rollover. For the most part, this data is never changed but merely augmented. You don't want to delete a customer's old address by overwriting it. Rather, you want to add to their information that, as-of a new date, their address has changed.

Archiving Retired Data

When this pattern is used, all data is append-only, versioned, retired-to-read-only, archived, and eventually discarded by deleting it from the archive. This is true for reference data, collaborations, activities, and resources.

As I've said before, accountants don't use erasers or they go to jail.

In the autonomous computing pattern, data is appended to collaborations, activities, and resources. Data is never updated. Only after retirement is it archived and eventually deleted.

4.4. Nested Composition of Fiefdoms

Big complex fiefdoms are broken into sub-fiefdoms. It is not an exaggeration to say that trust across departments in a large enterprise may be lower than the trust one department has for its well-known customers. Also, Conway's Law says that the system you build will be shaped by the internal organization and communication paths of the organization that built it.

To bound that trust across the internal pieces of a fiefdom, subsystems are frequently implemented as internal fiefdoms connected by their own internal collaborations.

Internally, the activities and resources may create their own interfaces using collaborations. Using collaborations and their messaging structure means the completion of each step of the work can be asynchronous and maybe collaborate with outside suppliers at other companies. When a large order is placed, coordination of delivery from suppliers may be needed. This asynchronous workflow is nested within the fiefdom's work for its customers.

Sometimes, a relatively small customer request may launch a chain reaction of side-effects. If a simple hotel reservation causes the occupancy for one of the nights to cross a threshold, the hotel may plan for a busier restaurant that night. Additional staff and food may be required. That, in turn, may launch outgoing collaborations to the grocery vendor. It is important, though, that this grocery order not be directly tied to the single activity of this hotel reservation. If I cancel my trip, that will not necessarily cancel the groceries. See Side Effects, Front and Center!

5. Scale Agnostic Fiefdoms and Emissaries

Let's consider an enhancement to the original autonomous computing pattern by considering how fiefdoms and their emissaries can be "almost infinitely scalable". This superimposes a programming pattern I originally proposed in 2007 to support **scale agnostic applications** within a **scale aware platform**. See my 2007 paper Life Beyond Distributed Transactions: An Apostate's Opinion. Let's consider the application of these concepts to the internal implementations of fiefdoms, emissaries, and how to leverage collaborations to make this happen.

First, let's briefly summarize the concepts in Life Beyond Distributed Transactions by looking at **scale agnostic entities**. This shows us how scale can be supported in the **scale aware platform** with sharding, messaging, and placement of these entities. The implications on indexing and analytics are considered. *Here, I'm going to use the word "bubble" in lieu of the word "entity".*

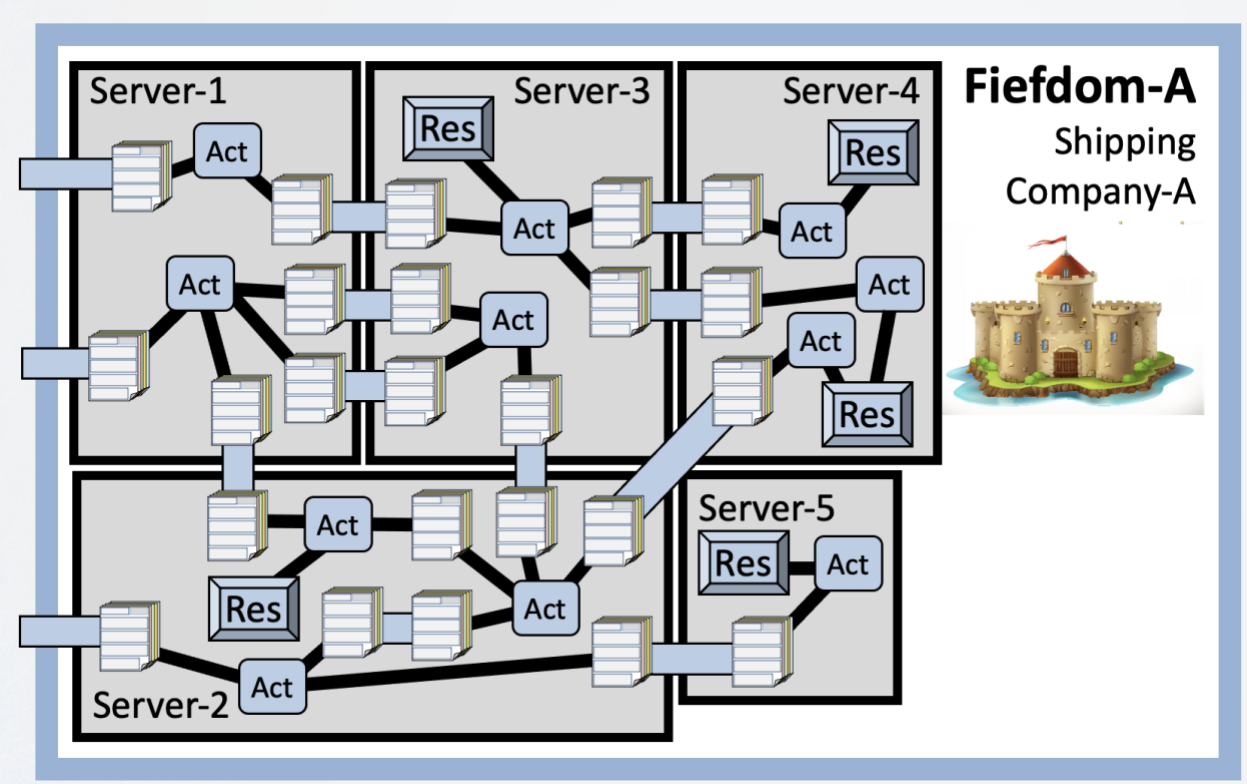
Moving ahead, we apply these concepts first to **scalable emissaries**. Where do the emissaries live? In autonomous partners? If so, how do we know they have long life and how can we deal with their possible failure? What about emissaries in smart devices? What about within web servers? Emissaries have lots of ways they can be built and built for scale.

What about **scalable fiefdoms**? How can we build collaborations that scale in an almost infinite fashion? Can we implement activities at scale and connect them to the activities and resources they need? How can resources scale and talk to the stuff they need? What about reference data, its creation, use and eventual retirement? Can that scale? Finally, we reiterate that all of these data and computational categories can scale while managing their lifecycle towards retirement.

5.1. Scale Agnostic Entities Connected with Collaborations

In Life Beyond Distributed Transactions, I explore how to build almost infinitely scalable applications while only using local transactions possibly on thousands and thousands of servers. I postulated that scalable apps work by creating **entities with unique keys**. Here, I'll call these bubbles. These bubbles are each small enough to always reside on a single server while the massive collection of millions or billions of them may require 10s or 100s of thousands of servers.

Each bubble has a unique key and transactions are constrained to always work within a single bubble. While multiple bubbles may reside on a single server, the **scale agnostic application** cannot ever perceive that they are updated atomically. Instead, work flows across bubbles via messages over collaborations. By placing activities and resources within bounded bubbles connected by collaborations, we can scale the solution while the application programmers are agnostic to scale. Collaborations connect multiple activities with bi-directional, full-duplex, in-order, and exactly-once messaging. Even when placed in separate bubbles.



Above, we've spoken of collaborations connecting fiefdoms and emissaries as they run asynchronously and possibly offline. Here, we are applying the same collaboration paradigm to connect the activities and resources *within* an **almost infinitely scalable fiefdom**.

In my 2007 CIDR paper, Life Beyond Distributed Transactions: An Apostate's Opinion, I spoke of connecting entities with messaging. Here, these are called "Bubbles" and the messaging pattern is more formalized as collaborations.

Each activity and each resource lives within exactly one bubble. The bubble (just like an entity) is a small and bounded amount of data and the computation used to encapsulate the data.

Bubbles are connected with shared collaborations.

Collaborations are shared across multiple collaborating partners. Each collaboration is replicated across its partners.

Since a collaboration comprises the messages sent by one of the partners, replication is easy. If there are three partners, there's three sequential lists of messages sent on the collaboration. Replicating the collaboration means squirting the new messages to the other partner as needed. Each replica has a subset of the messages sent using the collaboration and given time and connectivity, all replicas are identical.

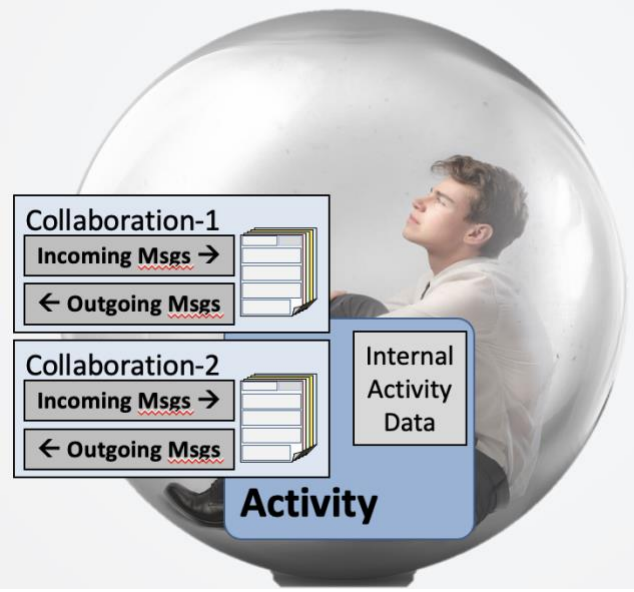
Collaborating bubbles may run on the same or different servers. They may be part of the same autonomous fiefdom or different fiefdoms. The collaboration pattern and the ability to replicate the set of messages sent empowers the pattern to run in many environments.

Scale is achieved because each bubble (i.e., one or a few activities possibly with a resource) may be dynamically moved to a different server when their old home gets too crowded.

One challenge involves location discovery and routing of the collaboration messages to the correct server. We had that problem when an emissary sent messages to a remote fiefdom. Here, we also have the problem *within* the almost infinitely scalable fiefdom. Fortunately, the message delivery guarantees are asynchronous. We can build mechanism to reroute messages within each collaboration when one of its collaborating activities or resources is moved.

5.2. Bubbles, Collaborations, and Activities

An activity is created in response to an incoming collaboration. It manages the state machine for the messages of that collaboration.



In addition, an activity may create new collaborations to do further work for the business. These collaborations have their own sequence of messages and set of business rules for managing the messages. If an activity has multiple collaborations, it deals with the interaction of the states of these collaborations.

For example, I may want to book a trip to three cities in Europe. The activity managing my travel arrangements may establish a number of collaborations to airlines, hotels, and car rental companies. The dates and locations of the flights may depend on the availability of hotels in the correct cities on the correct dates. Managing this may require many collaborations and the interactions of long-running arrangements across hotels, airlines, and car rental companies.

A single activity and its many collaborations may live in a single bubble. The state machine of the activity is captured as data describing the work. Each collaboration has a replica of the messages seen by the activity (both sent and received so far). Each of the other collaborators (e.g., an airline) will also have a replica of the messages in the collaboration.

Each replica for a single collaboration may be slightly behind on the messages it's received so far from another collaborators. Over time, the replicas of a collaboration catch up and are the same.

5.3. Bubbles, Collaborations, Activities, and Resources

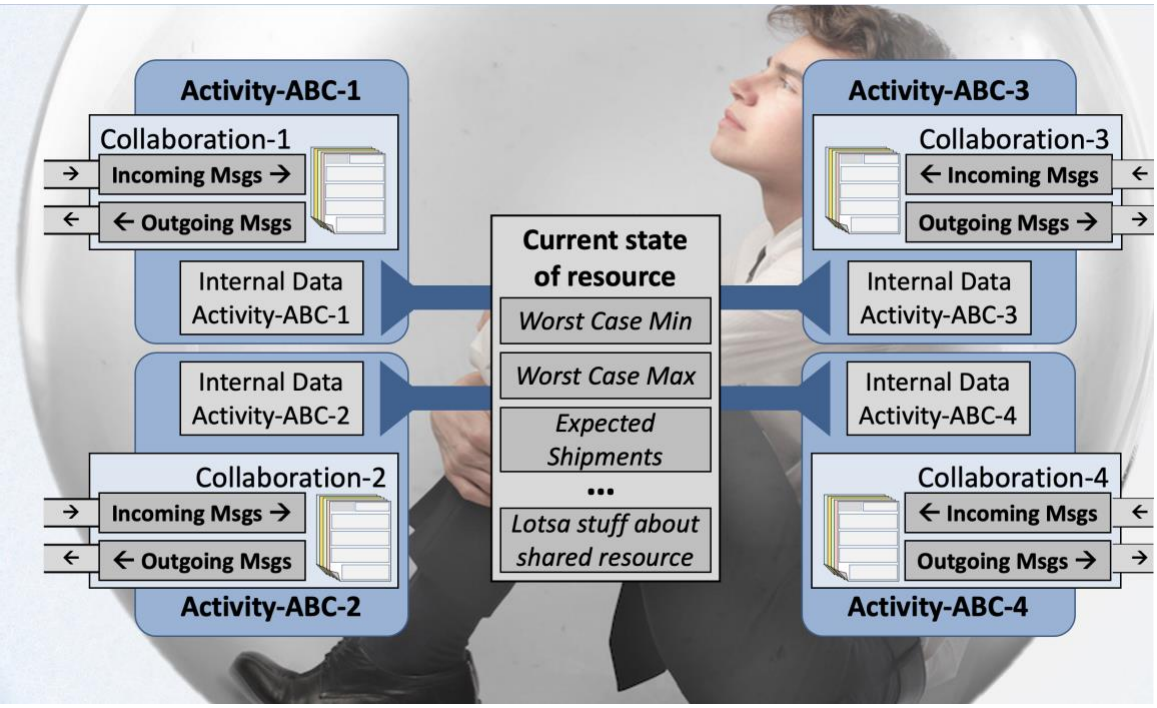
Each resource is fine-grained. There are many of them and they are relatively small. For hotel reservations, a single resource is used for the combination of many things:

- Specific hotel location
- Specific class of hotel room
- Specific date being reserved

Lots of resources are created and retired over time.

Each time some long-running work wants to reserve, cancel, or confirm access to some resource, there is an activity for that long-running work and its impact on the resource.

In my shopping cart for an online order accesses 5 different items, there will be many activities within the large online retailer. As this order interacts with the resource for each of the 5 different items being purchased, the impact on these 5 resources will evolve as shipments are scheduled and accomplished.



Each resource will have surrounding activities for each piece of ongoing work. The resource and its managing activities will live in a single bubble. Work arrives via collaborations and the activities dealing with incoming and outgoing messages.

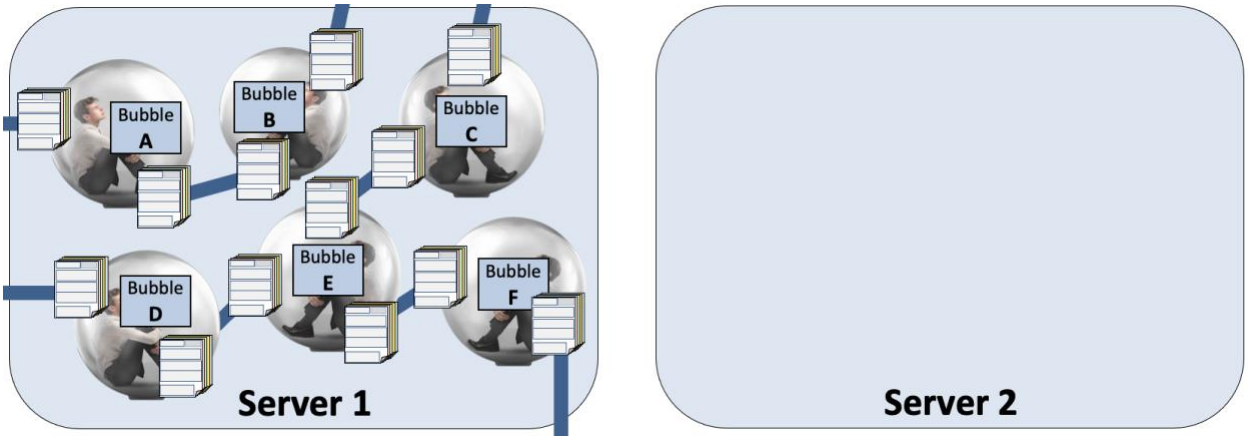
5.4. Moving Bubbles for Scale

Each bubble is a relatively small amount of data that lives on a single server at a time. Since it is small enough, the system can guarantee transactional execution of the work within a bubble. Transactions are invoked to handle either incoming messages on a collaboration or a timeout due to the lack of an incoming message.

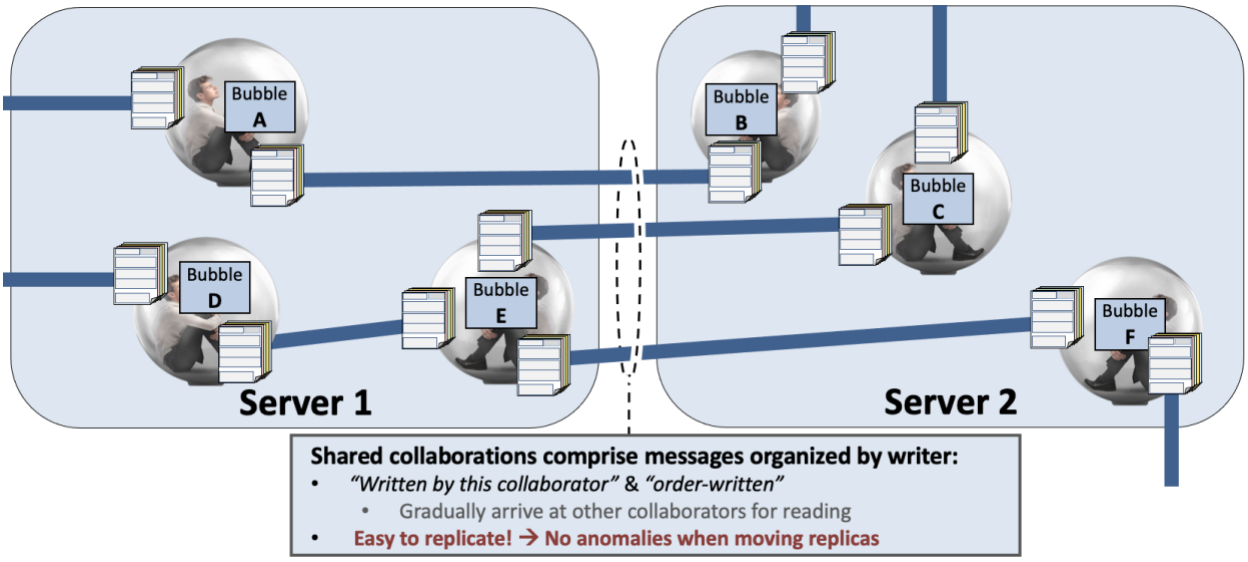
When a server has too much stuff, bubbles can be relocated to allow more scaling. To relocate a bubble, there are four steps:

1. Freeze the processing of the bubble
2. Move the bubble's state (including local replicas of each collaboration and its messages)
3. Update the routing tables for messages within the collaborations needed by the bubble
4. Reactivate the bubble to process incoming messages and timeouts.

Here, we see Server-1 getting too fat with six bubbles inside of it while Server-2 is empty:



After moving three bubbles, these two servers have split the work:



5.5. Entities, Messaging, Sharding, and Scale

Within the almost infinitely scalable fiefdom, long-running work is managed using messaging over collaborations. Collaborations with their transactional state at each partner can guarantee exactly-once and in-order delivery message delivery on each collaboration. They assign order at the sender and enforce it within the collaboration's code and data at the other partner. This makes life MUCH easier for the application code within the activity.

When resources are used, having a dedicated activity for each long-running piece of work helps. The activity implements the state machine for the sequence of messages in the collaboration. It also can consider the state of the shared resources as it does its work.

Messages across collaborations coming into and out of fiefdoms have ordering guarantees. The underlying system can ensure an activity only transactionally processes each incoming message exactly once delivered in the order sent by the partner on the collaboration.

What you don't get at scale is transactionally correct indexing. Indexes must be created asynchronously and behave like web-scale search. They do not behave the way they do within a relational database. Similarly, analytics are more challenging at scale and have a somewhat different semantic. In general, scalable analytics can be performed by gathering snapshots from the activities and resources, shipping them into a big data analytics framework, and answer questions about the state of the system at a point-in-time in the past. It is also possible to design analytic frameworks providing more current but somewhat inaccurate analytic perspectives. Spreading everything across thousands of servers imposes constraints on indexing and analytics.

5.6. Scalable Fiefdoms and Long-Running Work

As mentioned above, a fiefdom is comprised of collaborations, activities, and resources. Activities and resources can scale as long as they don't share transactions with other things. Collaborations can be used to connect activities and resources within the scalable fiefdom.

If the re-partitioning of activities and resources can also ensure the collaborations are able to locate the new home for these entities, the system can scale while continuing to provide service.

6. Conclusion: Trust, Collaborations, and Data

Autonomous computing is a pattern comprising collaborations across fiefdoms and emissaries to support business work.

Collaborations provide bi-directional, full-duplex, and asynchronous messaging connecting fiefdoms to emissaries and/or fiefdoms to fiefdoms. The premise behind the autonomous computing pattern is that each piece of the pattern, fiefdom or emissary, only interacts via messaging over prescribed and managed collaborations.

The same composition using collaborations can be used to provide almost-infinite-scaling of both emissaries and fiefdoms.

I argue this has been going on for many years. Indeed, the same patterns using paper forms have been used for centuries. We've just been slow in recognizing the pattern and empowering its easy and consistent use for our applications.

Hopefully, we can launch a more vibrant discussion of the autonomous computing pattern and how more support for it can be built. This can ease the complexity foisted on folks looking to solve business problems with computers letting them focus more on their business and less on hooking stuff together.