

# 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](mailto: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<sup>1</sup>. 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...

---

<sup>1</sup> 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.

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

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.

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...<sup>2</sup>

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<sup>3</sup> 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

<sup>2</sup> 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.

<sup>3</sup> Scaling at  $N \log N$  for some big log would be really nice...

message delivery inconsistencies foisted upon the innocent application developer as they attempt to build scalable solutions to business problems.

## Assumptions

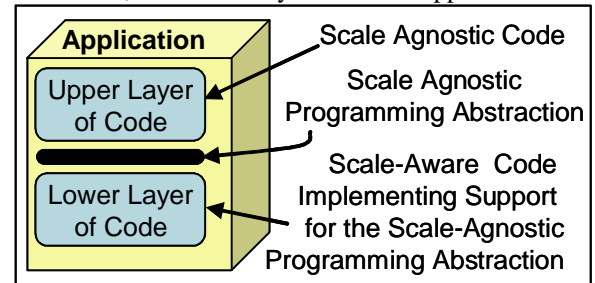
Let's start out with three assumptions which are asserted and not justified. We simply assume these are true based on experience.

- 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<sup>4</sup>.

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.

<sup>4</sup> Google's MapReduce is an example of a scale-agnostic programming abstraction.















