Design and Evaluation of a Reliable and Scalable Communication Framework

Craig Chasseur[1], Zuyu Zhang[2], and Jignesh M. Patel[2]
[1]*Google Inc.* [2]*University of Wisconsin–Madison*

Submission Type: Research

Abstract

Communication is a critical concern in any scalable distributed data processing system. While a number of communications frameworks have been proposed in the past, an evaluation of these approaches from the perspective of their features and performance is missing. In this paper, we address this gap. We also present Transactional Message Buses (TMBs), which build on ideas proposed in previous frameworks, but bring together a set of features that are desirable in a single communication layer to build systems that need to deal with aspects such as node failures and stragglers. The TMB architecture is highly modular, and allows different components implementing features like persistence and recovery, network transparency, and high availability to be combined freely, while still presenting exactly the same interface with the same well-defined semantics and guarantees regarding ordering to clients. We compare TMBs with existing approaches, consider a number of different implementations of TMBs, and present empirical experimental results. The TMB source code is available under the Apache open source license at `https://github.com/apache/incubator-quickstep/tree/master/third_party/tmb`.

1 Introduction

A crucial component of any scalable distributed system is a communication fabric that allows different actors in the system to communicate with each other. For example, a distributed search engine needs to send queries to different workers (each in charge of a distinct partition of the data) in the system, and then collect these results. A distributed data processing system needs a query coordinator to send sub-queries to different workers in the system and aggregate results.

In early distributed systems, such communication was often ad-hoc and not cleanly abstracted into a framework, which made programming and reasoning about concurrent behavior difficult and error prone, especially in the presence of unreliable components and networks. These challenges motivated the development of reusable general-purpose communications systems. Surprisingly, there has not been a systematic evaluation of such communication frameworks. We address this limitation in this paper. We also recognize that there are many good ideas and features in existing frameworks, but there isn't a single system that provides a comprehensive set of desirable features. Thus, we introduce a new communication framework called a Transactional Message Bus, or TMB. Table 1 compares the TMB's feature set with that of popular communication frameworks, including Apache ActiveMQ [2] (a message broker implementing the Java Message Service [20]), the Spread Toolkit [30] (a virtually synchronous messaging framework), Apache Kafka [4] (a distributed, partitioned message logging service), Akka [34] (a Java-based toolkit for building message-driven distributed applications), Amazon Simple Queue Service [1] (a cloud-based persistent queue service), and ZeroMQ [23] (a lightweight embedded broker-free messaging library). None of these systems have the *entire* ensemble of features present in a TMB.

The second contribution of this paper is to cleanly define the semantics of TMBs (see Section 3). TMBs are *transactional* in that the sending and receiving of asynchronous messages are ACID transactions with guaranteed delivery, data persistence and recovery support, and a consistent, deterministic set of semantics for addressing and ordering messages based on the well-known model of virtual synchrony [7] with some extensions. Although the message-level transactions provided by the TMB do not automatically translate to application-level ACID transactions in a distributed system, consistent and reliable messaging semantics can make it easier to implement application-level transactions.

Our last contribution is the design and evaluation of a "pluggable" modular software architecture for TMBs that allows components providing transaction management, durable persistent storage for messages, and network transparency to be freely combined into a complete TMB stack. Inspired by the observation that communica-

Feature	TMB	ActiveMQ	Spread Toolkit	Kafka	Akka	Amazon SQS	ZeroMQ
Point-to-point Messaging	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Group Messaging	Yes	Yes	Yes	Limited▲	Limited*	No	Limited*
Guaranteed Delivery	Yes	Optional	Yes	Yes	No	Yes	No
Persistent Queues	Yes	Optional	No	Yes	Optional	Yes	No
Queryable Queues	Yes	Yes	No	Yes	No	No	No
Virtual Synchrony	Yes	Optional✣	Yes	No	No	No	No
Robust to Client Failures	Yes	Yes	Partial♦	Yes	Yes	Yes	No
Robust to Service Failures (highly available)	Yes	Optional❍	Yes	Yes	N/A	Yes	N/A
Tied Messages	Yes	No	No	No	No	No	No
Priority & Deadlines	Yes	Yes	No	No	Deadlines Only	Deadlines Only	No
Message Size Limit	None	None	100 KB	Varies	Varies	256 KB	None

▲ Publish subscribe or "consumer group" that routes each message to single recipient only.
* Publish-Subscribe only.
✣ Virtual synchrony only available in single-broker or master-slave configuration.
♦ Messages can be lost if a client crashes during receive and later recovers.
❍ Highly available only in multi-broker or failover configuration.

Table 1: Messaging Framework Feature Comparison

tion between workflows via persistent queues is a database problem [18], we have implemented these TMB components using a variety of relational and NoSQL database systems (SQLite, VoltDB, Zookeeper, and LevelDB). We have also implemented our own lightweight "native" components at each level of the stack that do not depend on any third-party database. A key contribution of this paper is considering how a full-featured messaging framework can be built from various transaction processing systems, and what implications different approaches to transaction processing have for the reliability, performance, and scalability of messaging.

2 Related Work

One of the earliest abstraction for communication is the remote procedure call [10] (RPC), which allows invoking code on remote machines. More recent methods include Message Passing Interface (MPI) [19] to program distributed-memory systems, and CORBA [21, 36] to access "distributed objects." These methods cover some, but not all, features in TMBs. Notably, the notion of reliability of RPC has been addressed by systems like ISIS [8], which introduces the "virtually synchronous" execution model that is the basis for the TMB's semantics. To ease application programming, reliability and consistency features have been added to CORBA [29] – the TMB is based on a similar design philosophy.

Virtual synchrony provides distributed processes the illusion of a serial, synchronous sequence of events that occur in the same order for every process. The key aspects of virtual synchrony are address expansion, and delivery atomicity and order [7]. Address expansion means that sending a multicast message to a named group of processes requires all participants to have the same consistent view of group membership when the message is sent and when it is delivered. Delivery atomicity requires that all of the recipients of a multicast message will eventually receive or (only if the sender fails) none do. Finally, delivery ordering may be either causal or absolute. Causal ordering merely requires that for any two messages sent by the same sender to the same (or overlapping) receiver(s), the messages are received in the same order that they are sent (this corresponds to Lamport's definition of the causal happens-before relation [24]). Absolute ordering extends this requirement by imposing a total order on the receipt of messages for all processes in a group. By default, TMBs provide virtual synchrony with at least causal message ordering. TMB implementations that are based on relational database management systems (DBMSs) also provide absolute ordering. TMBs also provide mechanisms whereby applications can intentionally violate causal ordering so that, for instance, higher-priority messages arrive before lower-priority ones.

Message-Oriened Middleware (MOM) is a major industry [13], with various products [1, 2, 22, 27, 28], and standards like the Java Message Service [20]. MOM systems provide an asynchronous messaging service for cooperating applications. The MOM manages a queue of incoming messages on behalf of each client, and the acts of sending (enqueueing) and receiving (dequeueing) a message are decoupled and asynchronous. The basic MOM model has some limitations that have inspired extensions to integrate messaging with distributed ACID transactions [26, 33] and to integrate enforcement of logical conditions on message delivery into the MOM itself [32]. A key focus in this body of work is to "push down" features like at-least-once message delivery, deterministic ordering, and failure recovery to the communications layer to

make it easier to engineer distributed applications. Systems like Horus [35] have even made such functionality "pluggable" so that applications can select for themselves which features they need from the messaging layer. We have taken many cues from this body of work in developing the TMB, which also provides point-to-point and group-oriented messaging that we have integrated with strong reliability, consistency, and ordering guarantees as detailed in Section 3.

A big inspiration for the TMB was work realizing that message queues are themselves transaction-processing DBMSs [18]. Some MOM advocates countered that existing DBMS products were too "heavyweight" and slow for messaging applications [5]. However, as we show in this paper, modern DBMSs are more than up to the challenge.

The integration of transaction processing (TP) monitors and DBMSs also lead researchers to realize that to build a truly reliable TP system requires a persistent, recoverable system for request queues [6]. Reliable queueing of requests has been integrated into commercial DBMS products as part of a service-oriented database architecture [11]. TMBs owe much to lessons learned from persistent queues and generalize the concept to many different styles of asynchronous communication suitable for diverse application domains. TMBs also include built-in features such as tied messages, group messaging, and virtual synchrony which are crucial for modern distributed applications; thus, making writing distributed applications with TMB much easier.

3 TMB Semantics

In this section, we develop the TMB semantics including reliable and consistent message delivery.

3.1 Example

To illustrate the TMB API, consider a simple distributed search application. This application consists of several servers, each of which contains a different partition of some text data. Clients can search the data by sending a request to each server and collecting all of their responses, using the following TMB pseudo-code:

```
search(keyword):
  for server in search_servers:
    tmb.Send(server, SearchRequest(keyword))
  responses = tmb.Receive()
  return concatenate(responses)
```

The servers run the following message-driven loop:

```
loop:
  request = tmb.Receive()
  matches = find request.keyword in local_data
  tmb.Send(request.sender,
           SearchResponse(matches))
```

The server-side loop blocks until a message is available, then proceeds to search its local partition of the data, and send results back to the client that made the request. Sending and receiving a message are decoupled and asynchronous, so clients and servers don't have to wait for each other, but the guaranteed reliable delivery and abstract addressing provided by the TMB means that our application code doesn't need to worry about lost messages or retry loops. Thus, writing the application becomes simpler given the TMB abstraction.

3.2 TMB API

The calls in the TMB API are as follows:

- `Connect()`/`Disconnect()`: Connects a "client" (i.e. some actor using the TMB) to the bus so it can start sending and receiving messages, and permanently disconnects it, respectively. The `Connect()` call returns a unique ID that the client uses to identify itself to all other API calls.
- `RegisterClientAsSender()`/ `RegisterClientAsReceiver()`: Informs the TMB that a client is capable of sending or receiving a certain type of message. TMBs support sending any number of different application-defined classes of messages, which is discussed in detail in Section 3.4.
- `Send()`: Send a message to one or more other clients of the TMB. The arguments to this call include the message itself (tagged with a type identifier) and an address which specifies recipients (addressing is detailed in Section 3.5.1). Other optional arguments allow a sender to use additional messaging features that are described in Section 3.7.
- `Receive()`: Receive pending messages. This method is available in both a blocking version that waits until at least 1 message is available and a non-blocking version that returns immediately if no messages are pending for a client. These methods can be used to receive messages one at a time, or to get multiple messages in a batch.
- `DeleteMessages()`: Erase one or more received messages from the TMB. By default, `Receive()` does not erase messages as they are received, so that if a client fails or experiences some error, it can recover and not lose any messages. An explicit call to `DeleteMessages()` can be issued when a client has finished processing a message and no longer needs the TMB to retain it.
- `CancelMessages()`: Cancel a previously sent message, preventing any client from receiving it in the future. This call can be made by the client that originally sent a message, or by any of several clients that receive the message. Cancellation is discussed in detail in Section 3.7.3.

Every TMB API call is implemented as an ACID transaction on the TMB's state (note that this does NOT automatically mean that applications running on the TMB are transactional, but makes it easier to reason about concurrency and ordering). Below, we discuss the semantics of these transactions.

3.3 Clients

A client is an abstract entity that sends and receives messages using a TMB. Clients may be independent threads in a parallel program, independent processes running on separate machines in a distributed setting, or some other application-specific entity (for example, nodes in a graph-oriented processing model like Bulk-Synchronous Parallel). A client registers itself with the TMB by calling `Connect()`, which returns a globally unique identifier that the client uses to identify itself for any other call to the TMB API.

A TMB "remembers" a client and retains any metadata and pending messages until that client explicitly disconnects by invoking the `Disconnect()` API call. This means that even if a client fails and later recovers (for instance as a result of crash or hardware failure), no messages are lost, and other active clients may still receive messages that the client sent before failing, as well as send messages which a failed client can receive once it recovers. Thus the TMB provides highly available messaging even though clients may be unreliable.

3.4 Messages

Messages are the basic unit of communication between clients. There are no restrictions on the content of messages. A message is simply an arbitrary sequence of bytes that are opaque to the TMB itself. This abstraction allows virtually any serializable data structure to be a message, including text strings, flat programming language variables and structures, or any of several popular interoperable formats for structured or semi-structured data, such as JSON, XML, or Protocol Buffers.

Applications may use a TMB to send many different "types" of messages, and different clients may be capable of sending or receiving only certain types of messages. Each message has a "message type" identifier that is specified by the sender. Clients can register as senders or receivers for a particular message type, and the TMB enforces the policy that a client may only send a message of a type for which it is registered as a sender, and that any explicitly-specified recipients of a message must be registered as a receiver of the type. These tests are performed immediately as part of the `Send()` transaction, aborting and returning an error code to the sender if the check fails.

Let us say that our distributed search application defines `SearchRequest` as message type 0 and `SearchResponse` as message type 1. A client connecting to the TMB for the first time has the following start-up procedure (the server's procedure is the same, with the message types reversed):

```
my_id = tmb.Connect()
tmb.RegisterClientAsSender(my_id, 0)   # Request
tmb.RegisterClientAsReceiver(my_id, 1) # Response
```

3.5 Sending Messages

The core purpose of the TMB is to deliver messages reliably, but asynchronously. Asynchronous delivery has positive implications for the performance and scalability of TMBs, since clients generally do not need to wait for each other, as well as for availability, since senders can still send messages to clients that have temporarily failed or are "lagging" (processing messages slowly).

To send a message, a client calls `Send()`, supplying the message (including its type identifier) and an address that specifies one or more clients to receive the message (addressing is described in Section 3.5.1). The TMB checks that the client is connected and registered as a sender of the specified message type, and that each explicitly specified recipient is registered as a receiver of the message type. If the attempted send operation violates any of these constraints, the transaction is aborted and an error code is returned to the client. On the other hand, if the checks are successful, then a copy of the message (plus some additional metadata, including the client ID of the sender and the timestamp at which the message was sent) is pushed on each receiver's *queue* of incoming messages. As with other operations, pushing a message on a client/receiver queue is an ACID transaction. Indeed, each of the four ACID properties is important to the correctness of the TMB implementation: enqueueing a message must be *atomic* so that no partial, garbled, or misordered messages appear, *consistent* so that clients always see a valid queue state and that messages appear in the correct order (see Section 3.7 for details on ordering), *isolated* so that multiple concurrent senders and the receiver itself do not interfere with each other when enqueueing or dequeueing messages, and *durable* so that once a message is sent it is guaranteed to eventually be received by a client so long as that client (or a replacement that is brought up for it after a failure) continues to receive messages.

3.5.1 Addressing Modes

A TMB provides two ways of specifying which clients should receive a message. The first is *explicit* addressing, where a sender simply specifies a list of unique client IDs that should receive the message. A client may know these client IDs as a result of out-of-band communication, or as a result of previous communication using the TMB

(recall that the ID of the client that sent a message is provided to each client that receives it). To validate an explicit address, the TMB checks that each specified client is connected and, if so, if it is registered as a receiver of the message's type.

The second mode of addressing is *implicit*, where the sender simply requests that a message be delivered to any client that is capable of receiving it. If no connected clients are capable of receiving the message, then an error code is returned to the sender, otherwise the send transaction proceeds as normal using the set of receivers for the message type as its list of recipients.

Both modes of addressing allow for more than one recipient to be specified for a message. If the sender specifies that a message should be *broadcast*, then a copy of the message is enqueued for every recipient. If the message is non-broadcast, then a single client is chosen from the set of possible recipients to receive the message.

Combining implicit addressing and broadcast allows clients in our distributed search example to "discover" and send a search request to all servers, as follows:

```
status = tmb.Send(ANY, BROADCAST,
                  SearchRequest(keyword))
if status == NORECEIVERS:
  return {empty}
```

The combination of implicit addressing and broadcast also enables publish-subscribe style messaging in a TMB by using a different message type ID for each channel.

3.6 Receiving Messages

Each client that is connected to a TMB instance has a persistent, transactionally-consistent queue of incoming messages. `Receive()` observes a consistent snapshot of the queue and retrieves messages from it. In order to amortize the cost of accessing the queue, a client may choose to receive a batch of messages all at once. Both blocking and non-blocking versions of `Receive()` are provided. A client that operates a purely message-driven main loop (like the server in our example) would prefer the blocking version, since it has nothing to do without any messages, and waiting for the blocking call to return is more efficient than polling the non-blocking version. On the other hand, a client may not wish to block waiting for messages while none are available, instead performing some other useful work. In that case, the client can use the non-blocking version that returns immediately if no incoming messages are currently queued.

3.6.1 Deleting Messages

When and how messages are removed from queues is an important consideration for the reliability of an application that uses a TMB. We could remove messages from queues when they are received as part of the same transaction. However, this approach can cause problems for some applications when the client fails. Consider receiving and immediately deleting a message from the TMB's durable store as soon as a client receives it. Next, assume that the client fails before it can actually process the message. Later, when the failed client restarts and is ready to receive messages, it will simply start processing new messages. The message(s) that it received, but did not process before the failure event, will seem to have "disappeared". Thus, a client that fails can cause a violation of the durability and guaranteed message delivery properties.

To address this problem, receiving and deleting messages are *separate* operations in a TMB. `Receive()` is a read-only transaction that retrieves a message from the appropriate queue, but leaves the message in-place. Once a client has processed a received message and handled it in the appropriate, application-specific way (including possibly sending out responses or other additional TMB messages), it then makes a separate explicit call using the `DeleteMessages()` API. This second call triggers a separate transaction that actually erases the message from the queue. A client that makes multiple `Receive()` calls without deleting messages will see the same messages again, thus ensuring at-least-once delivery of messages even when clients fails. TMBs optionally allow messages to be deleted as they are received (admitting the anomaly discussed above for applications that can tolerate it), but this is not the default behavior.

Let us say that our distributed search application requires 100% recall, but servers sometimes suffer temporary outages. We can make the application robust to such outages by writing the server's main loop as:

```
loop:
  request = tmb.Receive()
  matches = find request.keyword in local_data
  tmb.Send(request.sender, SearchResponse(matches))
  tmb.DeleteMessage(request);
```

Now, if a server fails after calling `Receive()`, but before calling `Send()`, it will reenter the loop when it recovers, receive the request again, and proceed normally. Note that there is a small window where the server could fail after calling `Send()`, but before calling `DeleteMessages()`, in which case the server would do its local search and send a response again. A client-supplied request ID can be added to the messages so that clients can discard such duplicate responses, achieving exactly-once semantics even in the presence of failures.

3.7 Additional Messaging Features

Thus far, we have treated messages as opaque, aside from their type. In this section we develop some additional, but optional, features of messages that can make TMBs more useful in various application settings.

3.7.1 Deadlines & Priority Levels

In some cases (especially in interactive applications), a sender may only want a message to be received within a certain time frame. In our distributed search example, if some servers are lagging, then it may be preferable to return partial results to the user within a limited time frame rather than to wait for all the servers to respond [14]. After that window has passed, any outstanding requests are obsolete, and processing them is a waste of time on servers that were already lagging.

In order to avoid doing redundant work, which compounds the problem of lagging, an optional expiration time can be specified with the `Send()` call. In the corresponding `Receive()` call, each message's expiration time is checked against the timestamp of the transaction, and expired messages are silently discarded. Expiration times also affect the order in which messages are received by clients. Messages that expire sooner are received before those that expire later (messages that have no expiration time are received last). Thus, the TMB prioritizes messages with an earlier deadline.

Applications can also exercise explicit control over the relative ordering of messages by having senders specify a priority level for each message. Message queues are ordered in descending order of priority, with ties broken using the earliest-deadline-first policy described above. In our example application, we might have multiple classes of clients sharing the same service. Some are interactive and highly sensitive to latency, while others are doing offline batch-processing. We can assign a higher priority to the interactive requests so that they are serviced first, with the long-running batch jobs proceeding when the servers are not busy. This approach generalizes to any number of priority levels.

3.7.2 Ordered Delivery & Streaming

The features described above affect the order in which messages are received, which requires treating the queue of incoming messages for each client as a priority queue. Still, unless a sender explicitly and intentionally changes the order in which messages it sends to a particular client should be received by specifying different priorities or deadlines, it is useful for a sequence of messages from a particular sender to a particular receiver to be received in the same order as they were sent, i.e. to provide virtual synchrony with causal ordering [7]. This semantics facilitates streaming of data via multiple discrete messages, making it easier to reason about messaging between concurrent clients [9]. This functionality is easily achieved by using the send timestamp attached to each message to order messages by their send time when priority and deadline are the same.

3.7.3 Message Cancellation & Tied Messages

Tied messages are a highly effective technique for dealing with latency variability in large-scale interactive web services [14]. A major source of variable latency in large-scale distributed services is queueing delay on servers. A tied message is a request which is sent to multiple servers that are able to process it. A tied message includes information about each of the target servers. Clients issue a tied request to two or more servers that are capable of servicing it. Once a copy of the message reaches the head of one of the servers' queue, that server will "cancel" the message for its peers, preventing them from receiving it and doing unnecessary redundant work, while still allowing the client to benefit from having its request serviced by the lowest-latency server.

Tied or cancellable messages are fully supported by TMBs. When a message is sent, the sender may choose to make it cancellable. For such messages, the TMB creates a cancellation "token" that has information to locate and delete copies of the message in each recipient's queue. The cancellation token is attached to each copy of the message, and a copy is also returned to the sender. The sender may cancel a message at any time using the token. Similarly, receiving clients that receive a cancellable message can cancel it, thus preventing their peers from receiving it in the future.

Note that there is a small window of time during which it is possible for a client to receive a message that has just been cancelled. This situation is not an error for the TMB, as cancellation is treated as an idempotent operation. Nevertheless, programmers using a TMB should be aware that tied messages do not guarantee only-once delivery and take steps to ensure that multiple clients receiving the same message do not cause an application-level error (e.g. by doing operations that are idempotent and adding the original message id to the response message, ensuring that operations that modify a shared application state are idempotent or, failing that, coordinating using a distributed locking or commit protocol).

Tied messages can improve the tail-latency of searches in our example. Suppose each partition of the data is replicated across multiple servers. The client can broadcast a cancellable message to multiple servers for each replica set as:

```
search(keyword):
  for server_set in partitions:
    token = tmb.Send(server_set,
                     BROADCAST,
                     CANCELLABLE,
                     EXPIRES(now + 100 ms),
                     SearchRequest(keyword))
  rspns = {}
  loop until all received OR 100 ms:
    rspns = concatenate(rspns, tmb.Receive())
  tmb.CancelMessage(token) # stop redundant work
  return rspns
```

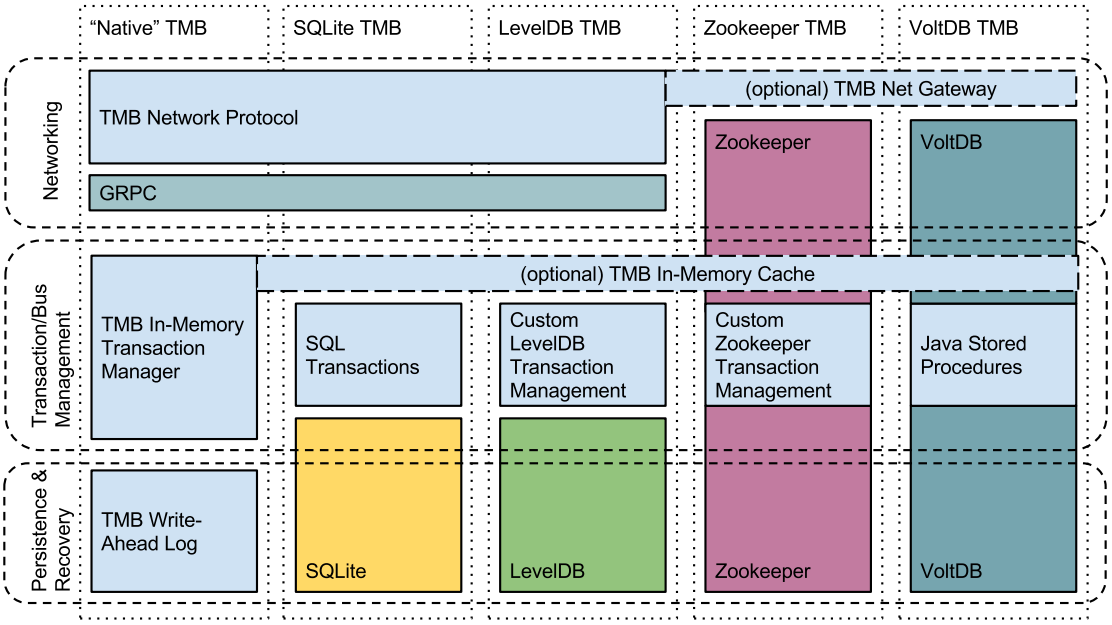

Figure 1: TMB Component Architecture

While the server code looks like:

```
loop:
  request = tmb.Receive()
  tmb.CancelMessage(request)
  match = find request.keyword in local_data
  tmb.Send(request.sender, SearchResponse(match))
```

The client benefits from having each request serviced by the first available server, but we limit load on servers by canceling tied requests when a server starts processing them, and after the client no longer wants more responses.

3.8 Summary of Logical TMB Structure

In this section we have developed the features and semantics of the Transactional Message Bus. Any TMB implementation consists of a shared globally consistent state, as well as per-client priority queues of incoming messages. The global state is the set of connected clients, and the set of message types that each connected client is capable of sending and receiving. All transactions modifying the global state have a serializable order. The per-client priority queues of incoming messages support *push* (i.e. send), *read* (i.e. receive), and *delete* (explicit removal or cancellation of messages) operations. The *push* and *delete* operations are atomic and serializable, and the read-only *read* operation observes a consistent snapshot of the queue.

4 TMB Implementations

In this section we discuss our experiences implementing the TMB as a modular service.

4.1 Modular TMB Architecture

The software architecture of a TMB implementation is divided into three tiers, as shown in Figure 1. At the heart of the TMB is a "Transaction/Bus Management" component that implements the full TMB semantics described in Section 3 and enforces the consistency guarantees described there. Below the transaction manager is a durable storage component responsible for storing TMB state persistently and recovering after a crash or other failure. Finally, there is a networking layer that allows TMB clients in different processes running on different machines to share a TMB instance and communicate with each other.

The components in each tier are abstracted to make them 'pluggable." For instance, if a TMB is only being used by clients on a single machine, there is no need for any networking component. Similarly, if there is no requirement for long-term durability of messages, the Native TMB in-memory transaction manager can be used without any persistent storage component as a high-performance in-process "pure memory" message bus.

4.2 Transaction/Bus Management Tier

For the bus-management core, we have implemented our own custom in-memory transaction manager, as well as pluggable interfaces for the following four transactional database systems: SQLite, LevelDB, Zookeeper, and VoltDB. Our in-memory transaction manager can also be used as an in-memory cache in front of these database systems, in effect using our high-performance in-memory transaction manager in combination with the durability and recovery that is provided by these four systems.

4.2.1 Custom In-Memory Transaction Manager

Here we describe our custom in-memory transaction manager, which can be combined with any of several options for persistence and recovery, or used alone as a "pure memory" message bus.

Data Structures The global set of connected clients is represented by a hash table that maps unique client IDs to per-client records. The per-client records, in turn, consist of a hash table of sendable message type IDs, a hash table of receivable message type IDs, and a priority queue of incoming messages (we have evaluated both max-heap and balanced binary tree-based priority queue implementations). There is also a secondary index that maps receivable message type IDs to client IDs to speed up the resolution of implicit addresses.

Concurrency Control Our initial transaction manager design used well-known concurrency primitives to control access to shared data structures. We found that the overhead associated with latching can cause severe performance issues (especially when there are many actors in a highly multi-threaded environment), so we designed a lock-free user space concurrency control mechanism that we call *HybridCC*, which borrows heavily from the read-copy-update (RCU) [25] paradigm, as well as rw-locks and reference-counting garbage collection. We omit details of the HybridCC implementation and its tuning for multi-socket NUMA use cases here, but a full description is available in the extended technical report [12].

The global hash-table of clients is protected by HybridCC. Connect and disconnect transactions modify the hash-table, while all other transactions observe snapshot isolation without blocking. Similarly, the per-client hash tables of sendable and receivable message types as well as the secondary receiver index hash-table are all protected by HybridCC instances.

Each client's queue of incoming messages is protected by a simple mutex, which is locked whenever any operation (*push*, *read*, or *delete*) is performed on the queue. To efficiently support the blocking version of the `Receive()` call, a *read* operation that sees an empty queue releases its lock on the mutex, and waits on a condition variable. A subsequent *push* operation that enqueues a message, which satisfies the minimum priority of the waiting *read* operation, signals the condition variable, thereby waking the reader.

4.2.2 SQLite Transactions

SQLite [31] is a popular embedded SQL database library that supports multi-statement ACID transactions. TMB transactions are implemented as SQL queries over state stored in five tables, with secondary indices to speed query evaluation when appropriate:

- **client** - Contains a row for each connected client.
- **sendable** - Contains one row for each message type sendable by each connected client.
- **receivable** - This table has the same schema as the **sendable** table, but for receivable message types.
- **message** - Contains columns for a unique serial message ID along with message contents and metadata.
- **queued_message** - Contains one row with a foreign-key reference to a row in the **message** table for each queued incoming message for each client.

4.2.3 LevelDB Transaction Management

LevelDB [16] is an embedded NoSQL key-value store. It supports *put*, *get*, and *delete* operations on individual key-value pairs, as well as iterators that allow seeking and scanning over keys in order. Multiple reads can be issued against the same consistent snapshot, and multiple write operations can be combined into a single atomic batch. We have built a minimal bus manager on the snapshot isolation and atomic commit features present in LevelDB. We use five different types of keys and mapped data structures in LevelDB that correspond closely to the five tables used in the SQLite implementation.

4.2.4 Zookeeper Transaction Management

Apache Zookeeper [3] is a distributed NoSQL data store with strong consistency guarantees. Zookeeper servers are usually configured as an "ensemble", with the service remaining available so long as a majority of the servers in the ensemble are up. Zookeeper servers synchronously log data changes to disk as part of all modifying operations, and a single Zookeeper server can be used without an ensemble if high availability in a cluster is not needed.

The Zookeeper data model is a tree of nodes not unlike a conventional filesystem, except that there is no distinction between files and directories. Our Zookeeper-based implementation has a directory structure that is broadly similar to the ordered key structure of the LevelDB implementation. The implementation of TMB transactions is also quite similar to the LevelDB implementation, but we had to handle and resolve anomalies that can arise from multiple reads in a transaction observing different versions of the data tree.

4.2.5 VoltDB Transactions

VoltDB [37] is a main-memory distributed SQL DBMS. The TMB implementation for VoltDB is similar to that of SQLite, with many of the same tables and transactions. VoltDB allows stored procedures to be written in Java, so we were able to embed some TMB logic directly in the database that in other cases required implementation in the TMB client library. To take advantage of VoltDB's

ability to efficiently execute transactions on different partitions of data in parallel, we denormalize the message contents and the metadata into the **queued_message** table, which is then partitioned on the receiver ID attribute. Stored procedures implementing the `Receive()` and the `DeleteMessages()` APIs are partitioned on client ID for parallel execution, as is a fast-path version of `Send()` for the common case of a single explicit recipient.

4.3 Persistence & Recovery Tier

In order for the TMB state to be persistent and recoverable after failures, we require a storage component that logs transactions durably and allows us to reconstruct consistent TMB state after a failure or interruption. For each of the four third-party databases we developed a TMB implementation that provides durable storage and recovery of committed transactions. We also developed our own minimal synchronous write-ahead log that can be used to replay a TMB's history and recover its state. Our native write-ahead log is simple, and uses POSIX atomic I/O syscalls for writes and the `fdatasync()` syscall to synchronously flush log records when committing.

4.3.1 Synchronous vs. Asynchronous Logging

By default, all TMB implementations synchronously flush logs to disk when committing a transaction so that data loss is never experienced in case of a crash. Some implementations do, however, allow logging to be asynchronous so that the operating system can buffer a number of log writes together before flushing them to disk, potentially allowing both lower latency for log writes and higher overall messaging throughput, with the caveat that some of the most recent messages may be lost in the event of a crash. Asynchronous logging is optional for our native write-ahead log, as well as the LevelDB and VoltDB implementations of the TMB. By default, we use synchronous logging for the strongest possible durability, but we do leave asynchronous logging as an option for users that are willing to accept the trade-off for increased performance. We conduct experiments in Section 5 comparing both styles of logging.

4.4 Networking Tier

Finally, we describe the networking tier, which is necessary for TMB clients running on different machines to transparently share a TMB and communicate with each other. There are two different approaches that we have evaluated in the networking tier. The first is a custom TMB network protocol that we developed on top of the GRPC cross-platform RPC framework [17]. In this protocol, there is a single TMB server that is responsible for running the TMB's transaction manager, and all client machines connect to the server. If the server crashes, clients must wait for it to become available again to continue messaging (TMB calls will time out or fail with a network error during this window, but clients can remain up and TMB state will be restored exactly as it was upon recovery). We note that a number of cloud-hosting services offer hot restart for VMs that crash, very quickly bringing up a replacement server connected to the same persistent disk. Although this is not truly uninterrupted service, it may give sufficiently high uptime for many applications.

The other approach to the networking tier is to leverage the built-in network transparency of an existing distributed database (in our prototypes, we have experimented with both Zookeeper and VoltDB). In this case, the TMB library on clients communicates directly with Zookeeper or VoltDB servers, and transaction management is handled in the database itself. Zookeeper servers are typically configured as an "ensemble" consisting of an odd number of machines, with the overall system remaining available so long as a majority of the servers are up and the network is not partitioned. VoltDB clusters have a user-tunable "K-safety" parameter, which causes each partition of data to be replicated across K + 1 different servers in the cluster, with replicas distributed so that any K servers can fail simultaneously and the cluster will remain fully available with all partitions online. This latter approach to networking has the advantage of high availability with zero interruptions in the face of individual server failures, although it may come at performance penalty since transactions must be applied at multiple replicas instead of a single server. Operational costs are also likely to be higher in this scenario, since multiple servers with uncorrelated failure domains must be kept running. An experimental evaluation comparing both networking approaches is contained in Section 5.4.

5 Experiments

In this section we present empirical results comparing the various TMB implementations.

5.1 Stress-Test Benchmark

To evaluate the performance of different TMB implementations, we devised a stress-test throughput benchmark. This benchmark starts a configurable number of sender and receiver threads, each of which connects to an TMB instance as a client. The sender threads repeatedly send messages as quickly as possible, randomly choosing one receiver for each. We measure the total aggregate throughput across all receivers in messages per second. We conducted experimental runs with each TMB imple-

mentation, varying the number of sender threads and measuring the impact on throughput.

We conducted experiments to measure both the intra-node and inter-node (i.e. scale-out) scalability of our TMB implementations. We benchmarked each TMB implementation on a multi-socket NUMA server with four Intel Xeon E7-4850 CPUs running at 2.0 GHz (each CPU has 10 cores and 20 hardware threads with 64 GB of directly attached memory), with a four-disk striped hardware RAID as persistent storage.

For our first round of experiments, we set the affinity mask of our benchmark executable so that it would run on only one CPU socket and access only local memory. We then conducted another round of experiments where we used all four sockets to measure NUMA scalability. When testing the Zookeeper and VoltDB implementations, the server process was run on the same machine.

To measure inter-node performance, we configured a cluster of dedicated virtual machine instances in Google Compute Engine [15]. We configured servers to run either the standalone TMB network protocol server or the underlying Zookeeper or VoltDB service with 8 Xeon CPU cores at 2.3 GHz, 30 GB of RAM, and a 128 GB SSD. When benchmarking the TMB network server, we used a single server. For experiments with Zookeeper and VoltDB we used three servers, meaning that the Zookeeper ensemble could tolerate the loss of one server and remain available, and we similarly configured VoltDB with a K-safety factor of 1. We configured a number of "application" nodes to run the benchmark with 4 Xeon CPU cores and 15 GB of RAM. We conducted experimental runs with 1, 2, 4, 8, and 16 application nodes, varying the number of threads running on each.

We also sought to compare the performance of the TMB against existing message-oriented middleware, specifically Apache ActiveMQ (a message broker for the Java Message Service) and the Spread Toolkit (a multicast distributed messaging service with virtual synchrony). We ported our stress-test benchmark to ActiveMQ and Spread, and ran it under the exact same cloud server configuration that we used to evaluate distributed TMB implementations (three 8-core servers running ActiveMQ brokers or Spread daemons, 1-16 quad-core application nodes running client threads). ActiveMQ was configured to use replicated LevelDB storage for persistent queues, with the three servers acting as a quorum with automatic failover for high availability. The three Spread daemons were configured as a single "segment" with safe, and fully atomic, multicast. Note that Spread does *not* support durable message queues (i.e. messages can be lost if daemons fail), so ActiveMQ and TMB are somewhat disadvantaged by their requirement to log messages durably in this pure performance comparison.

5.2 Distributed Search Application

We also developed a sample distributed search application using the TMB whose structure follows the example presented throughout Section 3, with the addition of a simple term frequency-inverse document frequency (TF-IDF) ranking function. A client submits a set of keywords for text search, and servers scan documents and return a hit list with frequency counts for each keyword in each matching document. The client then counts the total number of matching documents for each keyword to determine each keyword's inverse document frequency, and finally ranks all the matching documents according to TF-IDF.

We ran four search servers (matching our application node configuration above), each containing partitions of a large English plain text corpus with 44 partitions in total, each approximately 100 MB in size. Each data partition was replicated on two different servers, and we distributed the replicas so that each server contained a copy of 22 different partitions. We had a client submit a keyword search request to one server for each partition. We then simulated a straggler node by causing one of the servers to make four passes over the same data before responding. Finally, we had the client submit tied messages to both servers for each partition in an attempt to mitigate the performance impact of the straggler (servers cancel a request for their peer when they begin working).

We used the VoltDB-based TMB implementation for this experiment. All results below use the stress-test benchmark, except Section 5.6, which contains the results for the distributed search application.

5.3 Single-Node Performance

Figure 2 shows the relative performance of five TMB implementations when running on a single CPU socket (10 cores / 20 hyperthreads). Recall that these implementations were described in Sections 4.2.1 through 4.2.5. In Figure 2, the label "Native" indicates the use of the TMB in-memory transaction manager in combination with the TMB write-ahead log. The native log, LevelDB, and VoltDB are all configured to use synchronous logging for the strongest possible durability.

The first finding from this experiment is that the Native, LevelDB, and VoltDB implementations vastly outperform and out-scale the SQLite and Zookeeper implementations. Both LevelDB and VoltDB scale throughput well with additional threads before eventually leveling out when the number of sender threads far exceeds the number of hardware threads. The Native implementation is I/O-bound and has flatter performance, but it should be noted that it achieves the highest throughput when the number of threads is equal to the number of physical CPU cores (10), and achieves throughput only 9.5% below Lev-

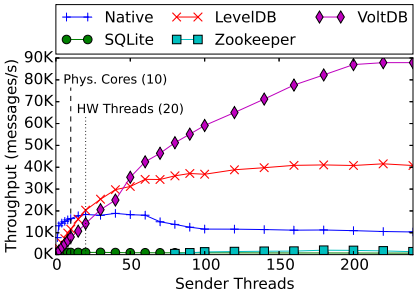

Figure 2: Single Socket TMB Performance (Strong Durability)

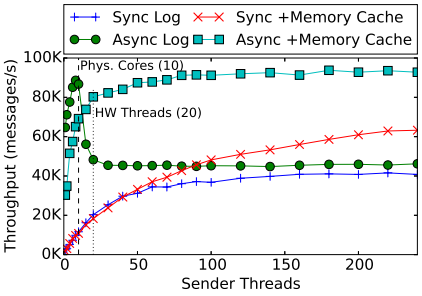

Figure 3: Effect of Memory Cache (LevelDB - 1 Socket)

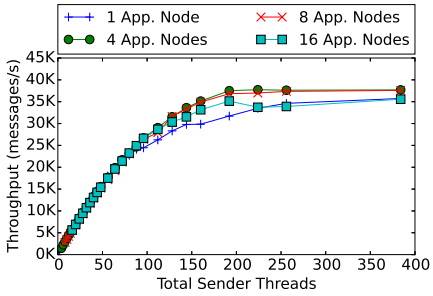

Figure 4: Cluster Scale Out (TMB Net Server with LevelDB Storage)

elDB's peak of 20318 messages/s when the number of threads is equal to the number of hardware threads (20). VoltDB ultimately scales up to a higher peak throughput when threads are heavily oversubscribed to cores.

We also examined the performance impact of optional asynchronous logging for the Native, LevelDB, and VoltDB TMBs. All three achieved higher messaging throughput, with the Native TMB hitting 5X higher peak throughput when the number of sender threads is equal to the number of CPU cores (10).

In Figure 3, we examine the performance impact of using the TMB in-memory transaction manager as a cache as described in Section 4.2.1, using LevelDB as the underlying storage engine. Using the custom TMB transaction manager as an in-memory cache improves the scalability of the LevelDB-based TMB considerably, regardless of whether synchronous logging is used. In the asynchronous case, the TMB cache eliminates the need for any explicit snapshotting as well as any read contention, reducing the interaction with LevelDB to a series of small point writes, with average throughput of up to 93764 messages/s with 180 sender threads. In the synchronous case, the TMB cache allows the message bus to continue scaling up to 63272 messages/s with 240 sender threads, a 55% improvement over sync logging without the cache.

We also tested the TMB in-memory cache with the SQLite, Zookeeper, and VoltDB-based TMBs. The cache resulted in only a slight improvement for SQLite (calls would still lock to serialize transactions), and had little impact on VoltDB (this is to be expected, since VoltDB is already a main-memory database). The Zookeeper implementation experienced a 3.3X improvement in peak throughput with the cache, although the highest throughput achieved was still only 6685 messages/s.

We also conducted tests using all four CPUs in our NUMA server, which we summarize here (a more complete discussion is available in the extended technical report [12]). We found that, for all persistent TMB implementations, the throughput scale-up curve was very similar to the single-socket case (suggesting remote memory access is not a significant bottleneck relative to disk I/O), and using the TMB in-memory transaction manager as a cache continued to boost performance.

5.4 Cluster Scale-Out

Figure 4 shows the results of our experiment using the TMB network protocol server for communication in a cluster. The server uses the TMB in-memory transaction manager, and uses LevelDB (with synchronous logging) for persistent storage [1]. We connected 1, 2, 4, 8, or 16 application nodes to the TMB, and measured throughput with the stress-test benchmark. This graph shows a similar relationship between the number of sender threads and the message bus throughput regardless of the number of application nodes. This shows that messaging throughput is effectively network-agnostic, with the TMB delivering the same throughput for a wide range of cluster sizes. The data for 1 node and 16 nodes show throughput that is slightly diminished relative to the other cluster sizes. The 1 node case is limited by the load from heavily oversubscribing threads to CPU cores on the application node, while the 16 node case is limited by high CPU load and many open network connections on the server.

Figure 5 shows the results of a similar experiment for the distributed VoltDB TMB implementation. Recall from Section 4.4 that a distributed database like VoltDB allows for a different approach to networking, with clients communicating directly with VoltDB servers and transaction management handled in VoltDB itself. We ran VoltDB on three server nodes (a cluster with K-safety = 1). Once again, despite the different approach to the networking tier, we see that messaging throughput scales in an effectively network-agnostic fashion, with throughput slightly diminished when clients or servers are heavily loaded.

We also experimented with a variable number of VoltDB servers (fixing the number of application nodes at 8). Adding additional servers increases the average message throughput in the cluster for any number of sender

[1]We use LevelDB for the TMB Net Server's persistent storage, as it showed the best performance characteristics of the synchronous persistence options in the single-node experiments described in Section 5.3.

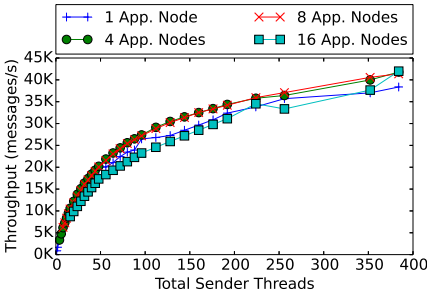

Figure 5: Cluster Scale Out (TMB on VoltDB)

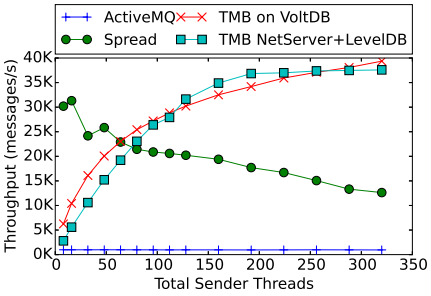

Figure 6: ActiveMQ vs. Spread vs. TMB (8 App Nodes Cluster)

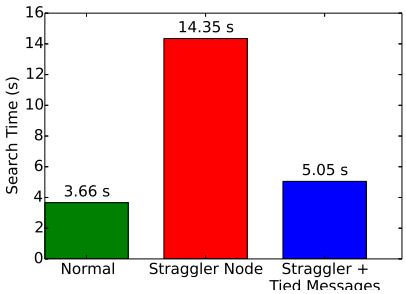

Figure 7: Search Latency In Distributed Search Application

threads, with the difference more pronounced for a higher number of threads. We do note that the throughput scale-up from adding more VoltDB servers is less than linear (for instance, with 384 sender threads, a configuration of 6 VoltDB servers achieved 27% higher throughput than 3 servers, while 9 servers achieved 42% higher throughput).

Finally, we evaluated the Zookeeper TMB implementation in the cluster environment. As with the single-node case, performance was underwhelming. The highest throughput was 4783 messages/s for 8 application nodes. Unlike the TMB network server and TMB on VoltDB, the relationship between threads and throughput was not cluster-agnostic, with smaller clusters of application nodes experiencing significantly lower throughput.

5.5 Comparison With ActiveMQ & Spread

Figure 6 shows the throughput of our stress-test benchmark using the TMB network server with LevelDB storage and TMB on VoltDB vs. the ActiveMQ message broker and the Spread Toolkit. With 8 application nodes, ActiveMQ's message throughput is in the range of 950 to 990 messages/s regardless of the number of sender threads. Spread achieves its highest throughput when there are few threads running on each node (30205 message/s with only a single sender and receiver thread on each VM), with performance diminishing due to contention as additional threads are added. In contrast, the two TMB implementations' throughput scales with additional threads, with throughput from 6.4X to 34X higher than ActiveMQ depending on the number of threads. Either TMB implementation achieves higher throughput than Spread above 64 sender threads (the number of cores in the cluster) and, unlike Spread, the TMB stores messages durably.

Both the TMB network protocol server and TMB on VoltDB have very similar throughput curves. The comparison is not entirely fair, however, as the TMB network server uses only a single machine, while TMB on VoltDB uses three. On the one hand, this means that operational costs for the TMB network server should be lower. On the other hand, TMB on VoltDB is more resilient to server failures, and can maintain high availability with zero downtime if one server fails.

These results show that the TMB design approach, leveraging either a custom-built lightweight transaction manager and network protocol or a high-performance distributed DBMS, compares favorably with leading purpose-built MOM systems.

5.6 Distributed Search Results

We ran our sample text search and ranking application using a cluster of four search servers each containing replicas of a large partitioned text data set as described in Section 5.2. The results for this experiment are shown in Figure 7. We ran a full-text keyword search on the unloaded servers, which completed in 3.66s. We then simulated a straggler node, which caused the completion time to increase to 14.35 s. Finally, we enabled cancellable tied messages, which allowed the search to complete in 5.05 s despite the slow performance of the straggler node. This demonstrates the effectiveness of tied messages in dealing with lagging and unreliable components.

6 Conclusions

In this paper, we have compared the features in a number of existing communication frameworks, and designed a new communication framework called the Transactional Message Bus that provides a combined richer set of features than existing approaches. We presented the semantics of the TMB, and designed a modular, pluggable multi-tier software architecture for TMB implementations. We have developed and evaluated a number of alternative TMB implementations using this architecture, some of which are "custom" and written from scratch and others which leverage the features of existing database systems. We also compared the performance of a TMB with the popular messaging frameworks Apache ActiveMQ and Spread. Our results show that TMBs achieve performance far higher than ActiveMQ and competitive with Spread, as well as a richer feature set than either.

References

[1] Amazon Web Services, Inc. Amazon Simple Queue Service. https://aws.amazon.com/sqs/, 2014.

[2] Apache Software Foundation. ActiveMQ. https://activemq.apache.org/, 2014.

[3] Apache Software Foundation. Apache Zookeeper. https://zookeeper.apache.org/, 2014.

[4] Apache Software Foundation. Apache Kafka: A high-throughput distributed messaging system. https://kafka.apache.org/, 2015.

[5] G. Banavar, T. D. Chandra, R. E. Strom, and D. C. Sturman. A case for message oriented middleware. In *DISC*, pages 1–18, 1999.

[6] P. A. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. In *SIGMOD*, pages 112–122, 1990.

[7] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, Aug. 1991.

[8] K. P. Birman. Replication and fault-tolerance in the isis system. In *SOSP*, pages 79–86, 1985.

[9] K. P. Birman. The process group approach to reliable distributed computing. *CACM*, 36(12):37–53, 1993.

[10] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. In *SOSP*, pages 3–, 1983.

[11] D. Campbell. Service oriented database architecture: App server-lite? In *SIGMOD*, pages 857–862, 2005.

[12] C. Chasseur and J. M. Patel. Design and Evaluation of a Reliable and Scalable Communication Paradigm (Extended Paper). http://quickstep.cs.wisc.edu/tmb/tmb-extended.pdf.

[13] E. Curry. Message-oriented middleware. *Middleware for communications*, pages 1–28, 2004.

[14] J. Dean and L. A. Barroso. The tail at scale. *CACM*, 56(2):74–80, 2013.

[15] Google Inc. Google Compute Engine. https://cloud.google.com/products/compute-engine/, 2014.

[16] Google Inc. LevelDB. https://code.google.com/p/leveldb/, 2014.

[17] Google Inc. GRPC. http://www.grpc.io/, 2015.

[18] J. Gray. Queues are databases. In *HPTS*, 1995.

[19] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22(6):789–828, Sept. 1996.

[20] Hapner, Mark and Burridge, Rich and Sharma, Rahul and Fialli, Joseph and Stout, Kate and Deakin, Nigel. Java Message Service. https://jcp.org/aboutJava/communityprocess/final/jsr343/index.html, 2013.

[21] M. Henning. The rise and fall of corba. *Queue*, 4(5):28–34, June 2006.

[22] IBM Corporation. WebSphere MQ. http://www-03.ibm.com/software/products/en/websphere-mq, 2014.

[23] iMatix Corporation. ZeroMQ. http://zeromq.org/, 2014.

[24] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[25] P. E. McKenney and J. Walpole. Introducing technology into the linux kernel: A case study. *SIGOPS*, 42(5):4–17, July 2008.

[26] C. Molina-Jimenez, S. Shrivastava, and N. Cook. Implementing business conversations with consistency guarantees using message-oriented middleware. In *EDOC*, pages 51–, 2007.

[27] Pivotal Software, Inc. RabbitMQ. http://www.rabbitmq.com/, 2014.

[28] Progress Software Corporation. Sonic MQ. http://www.progress.com/products/openedge/solutions/application-integration/aurea-sonic-mq, 2014.

[29] S. Ramani, K. S. Trivedi, and B. Dasarathy. Reliable messaging using the corba notification service. *Intl. Symp. on Distributed Objects and Applications*, 2001.

[30] Spread Concepts LLC. Spread Toolkit. http://www.spread.org/, 2014.

[31] SQLite Consortium. SQLite. https://sqlite.org/, 2014.

[32] S. Tai, T. Mikalsen, I. Rouvellou, and S. M. S. Jr. Conditional messaging: Extending reliable messaging with application conditions. *Intl. Conf. on Distributed Computing Systems*, page 123, 2002.

[33] S. Tai and I. Rouvellou. Strategies for integrating messaging and distributed object transactions. In *Intl. Conf. on Distributed Systems Platforms*, pages 308–330, 2000.

[34] Typesafe Inc. Akka. `http://akka.io/`, 2014.

[35] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *CACM*, 39(4):76–83, 1996.

[36] S. Vinoski. Corba: integrating diverse applications within distributed heterogeneous environments, 1997.

[37] VoltDB Inc. VoltDB. `http://voltdb.com/`, 2014.