A distributed database view of network tracking systems

Jason Yosinski[a] and Randy Paffenroth[a]

[a]Numerica Corporation, 4850 Hahns Peak Drive Suite 200, Loveland, CO 80538, USA

ABSTRACT

In distributed tracking systems, multiple non-collocated trackers cooperate to fuse local sensor data into a global track picture. Generating this global track picture at a central location is fairly straightforward, but the single point of failure and excessive bandwidth requirements introduced by centralized processing motivate the development of decentralized methods. In many decentralized tracking systems, trackers communicate with their peers via a lossy, bandwidth-limited network in which dropped, delayed, and out of order packets are typical.

Oftentimes the decentralized tracking problem is viewed as a local tracking problem with a networking twist; we believe this view can underestimate the network complexities to be overcome. Indeed, a subsequent 'oversight' layer is often introduced to detect and handle track inconsistencies arising from a lack of robustness to network conditions.

We instead pose the decentralized tracking problem as a distributed database problem, enabling us to draw inspiration from the vast extant literature on distributed databases. Using the two-phase commit algorithm, a well known technique for resolving transactions across a lossy network, we describe several ways in which one may build a distributed multiple hypothesis tracking system from the ground up to be robust to typical network intricacies. We pay particular attention to the dissimilar challenges presented by network track initiation vs. maintenance and suggest a hybrid system that balances speed and robustness by utilizing two-phase commit for only track initiation transactions. Finally, we present simulation results contrasting the performance of such a system with that of more traditional decentralized tracking implementations.

Keywords: distributed target tracking, distributed database, two-phase commit, initiation, SIAP

1. INTRODUCTION

Broadly stated, target tracking endeavors to estimate the *states* (e.g., positions and velocities) of one or more targets of interest using measurements provided by one or more sensors (e.g., RADAR or infrared). Oftentimes, multiple sensors are used to increase the accuracy, coverage, or responsiveness of the system. When many sensors are used, it is generally advantageous to fuse the data from all the sensors into a single set of tracks. The benefits of data fusion are twofold: the resulting track picture may be more accurate than that generated by a single sensor, and the track picture present on the various platforms may be synchronized between all sensors. In the air target tracking regime, this unified, fused result is often called a Single Integrated Air Picture, or SIAP.

The question at hand is then how to fuse the information from a diverse set of sensors. Fortunately, the stochastic state estimation algorithms used for a single sensor (maximum likelihood estimation, Kalman filters) are fairly easily extended to the multiple-sensor case. Extension to multiple sensors does introduce additional complexities, but these complexities are reasonably well understood; for example, sensor bias/drift may be mitigated through the application of a Schmidt-Kalman filter.[1]

The ease with which single-sensor algorithms may be applied to data from many sensors motivates what we refer to as the *Centralized Architecture*. In the Centralized Architecture, raw measurement data is sent from all sensors to a central location for processing. The central processor fuses the measurements into a single track picture and distributes this track picture back to each platform, if desired. By construction every platform will operate with an identical (though delayed by the network) copy of the track picture because a single set of tracks was generated at the central hub and broadcast to all the client platforms. Though the Centralized Architecture

Further author information: (Send correspondence to R.P.)
J.Y.: E-mail: jason.yosinski@numerica.us, Telephone: +1 970 461 2000 x 256
R.P.: E-mail: randy.paffenroth@numerica.us, Telephone: +1 970 461 2000 x 233

has an attractive simplicity, it suffers several fatal flaws. Sending all sensor data across a network often requires an infeasible amount of bandwidth. Centralized processing also introduces a single point of failure, which may be an unacceptable risk in certain applications.

The *Replicated Centralized Architecture* fixes the second of these two problems. As the name suggests, in this architecture each sensor broadcasts all of its data to all other sensors; each sensor then independently incorporates the data from all other sensors into its own network track picture. If any one sensor fails, the remaining sensors are able to continue processing, eliminating the single point of failure mode of the Centralized Architecture. The principle underlying the Replicated Centralized Architecture is fairly straightforward: if each platform runs the same algorithms on the same data, the results will be identical. However, although the Replicated Centralized Architecture successfully avoids the Centralized Architecture's single point of failure, it engenders its own set of issues. In the presence of network delays, different platforms on the network will receive measurement data in varying time orders; therefore, the standard algorithms used must be enhanced to accommodate data received in arbitrary order. In other words, we now require that the *same algorithms produce identical results given the same data in any order*. Unfortunately, this enhancement is nontrivial. Although much progress has been made in efficiently incorporating out of sequence measurements (OOSMs) into currently existing track states,[2,3] the problem of identical track initiation given OOSMs still remains. Finally, even if the OOSM problem could be completely solved, the network bandwidth required to transmit *all* local data to *all* remote platforms may still be prohibitive.

The standing issues with the Centralized Architecture and Replicated Centralized Architecture prompt the development of the *Decentralized Architecture*. In the Decentralized Architecture, each sensor operates semi-autonomously while cooperating with its peers to generate a consistent track picture, eliminating the Centralized Architecture's single point of failure. In studying algorithms used in such architectures, we begin by noting that the algorithms and issues that arise in decentralized network architectures decompose into two categories — *track initiation* and *track maintenance*.

Within the Decentralized Architecture, track maintenance — the process of updating a track's state using new sensor data — may employ several optimizations to decrease bandwidth. Rather than sending all measurements across the network, one may choose to send only *associated* measurements, or those measurements that have been determined to be associated with a pre-existing track. This can result in much lower bandwidth usage in the presence of a noisy sensor producing many false alarms. Also, one may choose not to even send *every* associated measurement but rather to compress several associated measurements into a single pseudo-measurement, known as a *tracklet*.[4,5] Both of these optimizations decrease the required bandwidth, but both also assume pre-existing tracks.

Track initiation, however, becomes more difficult in the Decentralized Architecture. When every sensor is treated as an equal peer, and all are allowed to initiate network tracks, a myriad of *race conditions* naturally arise. For example two platforms may nearly simultaneously attempt to initiate a new track, but due to processing and network delays, each will be unaware of the other's intentions, resulting in redundant tracks on the network. Several decentralized algorithms have been proposed to address these issues; herein we propose an alternate architecture that attempts to be simpler while also providing a solution which is *provably* correct in important special cases.

The concept behind the proposed architecture proceeds from the observation that both track initiation and maintenance can be viewed as *transactions* affecting some *system track database* that is distributed among peers on a network. From this point of view, there is a vast extant literature on distributed database systems that can be mined for possible application to distributed tracking problems.[6,7]

In the following sections, we identify pertinent details of a transaction-based tracking architecture and examine how they may be put to advantageous use in multi-sensor multi-target tracking problems. The core advantage of the proposed method is a *simpler and provably correct* handling of the race conditions inherent in system track initiation. The main disadvantage of the new method is the additional number of messages required for coordination of peer platforms, though the required additional messages are small enough that the total bandwidth may be smaller than in other decentralized architectures.

The current text begins by giving a rudimentary description of the necessary transactional data system concepts in Section 2 and in Section 3 extends these concepts to distributed databases. Section 4 provides a derivation of our track initiation infrastructure in terms of these standard database concepts, and Section 5 gives visual examples of the messages and state changes during two distributed transactions. We close with a set of simulation results in Section 6 and some conclusions in Section 7.

2. PERTINENT TRANSACTIONAL DATABASE CONCEPTS

Database systems have been extensively studied and documented.[6–8] In this section we define some basic concepts that will be important in our analysis of such systems and their applicability to distributed target tracking. The exposition here will be quite cursory, and will only suffice to motivate what follows. The citations above provide an entryway into the literature for those desiring more detail.

There are several defining characteristics of a database system. The first, and most commonly considered, is the *database model*. The database model defines the structure of the data and the set of operations that may be performed on it;[9] examples include hierarchical, relational, and object models. While considerations of database model are important, they will not be the focus of what follows; many models might be used as the basis of a tracking system database with various trade-offs. Instead we will focus on *database transactions* and *transactional databases*. Transactional databases include most practical database systems (i.e. there are transactional databases that use a variety of underlying models). This section describes several general properties, known as the ACID properties, of transactional databases; Section 3 addresses issues specific to distributed databases; and Section 4 details the creation of a network-centric tracking database.

In studying transactional databases, we begin by noting that a *transaction* is defined as a single logical operation on the database. We generally desire that transactions satisfy four properties: *atomicity*, *consistency*, *isolation*, and *durability* (known collectively as the ACID properties).[9] Transactional database implementations that maintain the ACID properties have been shown to lead to many desirable properties of the system as a whole.

We briefly investigate each of the ACID characteristics below. For clarity we also illustrate the importance of each property with the oft used example of a bank's database of customer records. Although this example is notably removed from the realm of tracking, we believe this separation may serve to illuminate the division between the tracking problem and the underlying database intuition. In Section 4 we will expound the connection between a distributed tracking system and the properties of its associated database.

Atomicity – The *atomicity* of a transaction refers to the transaction's existence as an indivisible unit: either it appears in the database in full, or it does not appear at all. It is particularly important that the end user be able to view a database transaction as an atomic operation, even though the transaction may include multiple internal database operations.

As an example of the importance of atomicity in database transactions, consider a bank's database of its customers' account balances. When a user wishes to transfer funds from one account to another, multiple database operations must be performed: both account balances must be read, one value incremented and the other decremented, and both account balances must be written. The absence of atomicity for this transaction clearly has undesirable consequences. If only one of the two writes occurs, then one account balance will not be updated, leaving the client either richer or poorer by the amount of the transfer. As a database designer, we would much rather the entire transaction fail than to have it proceed incompletely.

Consistency – *Consistency* requires that the transaction does not leave the database in a state where any of its *integrity constraints* are violated. The integrity constraints are chosen in a problem domain specific manner. To use our banking example, one may choose to define the constraint that no account be left with a negative balance; i.e., transactions that would result in a negative balance should fail. We will later see to defining our own constraints specific to network tracking.

Isolation – The *isolation* property demands that multiple transactions executed simultaneously do not impact each other. We will later see that this property is especially relevant to track initiation.

Consider two users each wishing to make a deposit to a common account, say \$10 for the first user and \$50 for the second user. Each of the two deposits will entail a separate transaction, which will involve reading the account balance into a local variable, adding the desired dollar amount to that local variable, and then writing the new value back to the database. If the two users' transactions are allowed to proceed in an interleaved manner (e.g., user1 reads, user2 reads, user1 writes, user2 writes), the two transactions may interfere. The interested reader may show that the account may end up being incremented by either \$10, \$50, or \$60, depending on the order in which the read and write operations occur.

Durability – *Durability* requires that changes to the database persist in the face of failure. Once a user is notified that a transaction has been committed, that transaction may not be later aborted or forgotten. This is an obviously desirable property whose difficulty to ensure depends on the particular failures the system must endure.

3. DISTRIBUTED DATABASES AND TWO-PHASE COMMIT

Enforcing the ACID properties on a local database system is difficult to accomplish, and it becomes even more so when the database is distributed across many computers on a network. Such a system is known as a *distributed database system*, or DDS. In practice databases are distributed for a variety of reasons. The database may be too large to implement on a single machine, or having a live remote backup may be desirable. For our purposes we require a distributed database because the data of interest is generated and used at spatially diverse locations.

To implement a distributed database in compliance with the ACID manifesto, we typically rely on a distributed locking procedure. The *two-phase commit* algorithm is a commonly used method for accomplishing this task. The two-phase commit protocol allows all peers in a distributed database to agree either to commit or to abort a transaction. The two phases of the algorithm consist of a *commit-request* phase in which a platform (the *initiator*) that wishes to commit a transaction attempts to prepare its peers on the network, and a *commit* phase in which the initiator completes the transaction by informing the peers of the transaction's success or failure. Here we follow other common derivations[10] to give a rough outline of the algorithm.

3.1 Phase 1 – commit-request

The commit-request phase begins when the initiator decides to initiate a network transaction. The initiator *locks* its database and broadcasts a *commit-request message* to all the peers on the network asking for their consent to commit a transaction. When we say that a peer locks its database, we mean that it temporarily restricts local read or write access to the database. Each peer's database must remain locked as long as the transaction is undetermined. If database reads or writes are allowed before the transaction is determined, then platforms may make inconsistent decisions based on different information.

When each peer receives the commit-request message, it decides whether or not to approve the transaction. Generally, the only reason a peer would choose not to approve a transaction is if it has sent out its own commit-request message which takes *precedence* over the remote commit-request. Based on its decision, the peer sends its vote (yes or no) back to the initiator. If it votes yes, it also locks its own database; if it votes no, it knows that the transaction will ultimately fail, so it is not necessary for it to lock its database.

The commit-request phase concludes when the initiator has received either *yes votes* from all peers or a *no vote* from any one peer. The top two diagrams in Figure 1 show the commit-request and vote messages that comprise the first phase of two-phase commit.

The attentive reader may suspect that hidden details lurk beneath the surface of the previously mentioned concept of *precedence*. Indeed, this is the case, and a brief discussion is in order. A key requirement of commit-request messages is that they are uniquely *serializable*. That is, given a set of such messages, each peer on the network must be able to impose a unique ordering upon them; moreover, this ordering must be agreed upon by all other peers. To determine precedence, platforms use the chosen serialization to compare the two messages and choose to accept the one with higher priority.

In many cases serializability of messages may be easily achieved through the use of message time stamps. Each platform annotates the commit request message with the time at which it was sent, and the serialization

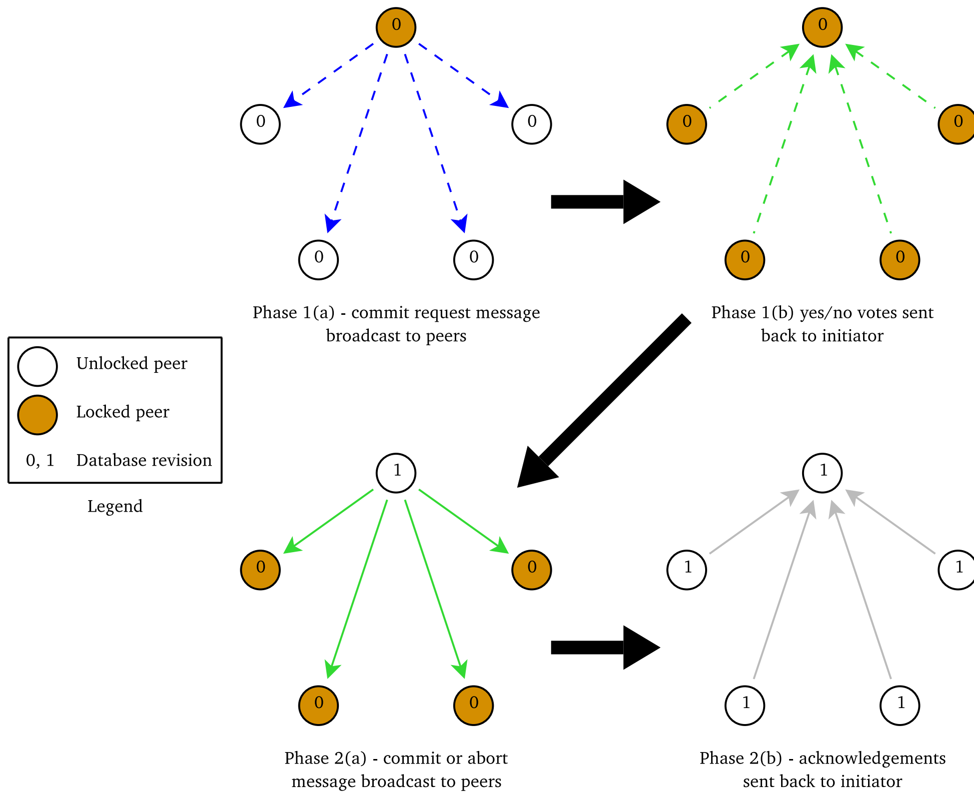


Figure 1. The messages passed between peers in the two-phase commit protocol. The database states (locked or unlocked) and revisions are depicted assuming the messages shown are still in transit.

rule is chosen such that earlier time stamps have priority over later ones, perhaps with platform IDs used to break rare ties.

At this juncture we note the difference between a serialization scheme that is *correct* and one that is *fair*. The serialization scheme using only platform IDs, for example, will be correct — all peers will agree on the ordering — but not very fair. Platforms with lower platform IDs will always have priority when attempting to commit transactions, which may lead to undesirable consequences, such as higher average system latency. By choosing to use message time stamps, we design the system to operate with minimum latency; the first platform to submit a transaction will ultimately succeed, and all other later transactions that interfere will fail.

From the distinction between the correctness and fairness of a serialization scheme comes an important result: accurate platform clock synchronization is not required for the correct operation of two-phase commit. Even if two peers have different clock biases, they can still agree on the relative order of time stamps. Of course, as the biases become very large, the system will become increasingly less *fair*, as the platform with the slowest clock will receive greater priority, but it will always still be *correct*, as the peers will be able to agree on the ordering.

3.2 Phase 2 – commit

Once the initiator has received either (a) yes votes from all peers or (b) a no vote from any peer, it is able to determine whether or not the transaction should succeed. If the initiator has received any no votes, it may immediately abort the transaction and send out *abort messages* to the peers. However, if it receives yes votes from all other platforms then the transaction is a success; the initiator may apply the transaction to its own database and broadcast the *commit message*.

When each peer receives either the commit or abort message from the initiator, it resolves the transaction by applying to it to its database or discarding the transaction, respectively; in either case it is free to unlock its database. Finally, each peer sends an *acknowledgment message* back to the initiator, confirming receipt of the commit or abort message. The bottom two diagrams in Figure 1 show the messages comprising the second phase of two-phase commit.

3.3 Caveats

Unfortunately, it is well known that the above algorithm is blocking.[8] For example, an initiator who sends out a commit-request blocks until all votes are received. To be precise, when we say "block" we mean that the transaction is not resolved until all votes are received; of course, other processes on the initiator and the peers are allowed to proceed. On the other hand, it is also well known that permanent blocking can be avoided through the use of timeouts. For example, a commit-request that does not receive timely votes can be resent (either to the same set of peers or a new set of peers if peer failure can be detected).

There are additional assumptions that need to be satisfied to make the algorithm fully functional, but most are rather mild in the present context (for example, those having to do with "stable storage"). One additional assumption that is important for target tracking problems is that all peers are known on the network. This is needed so that the votes may be tallied. There is an abundance of recent research[11,12] that addresses this issue as well as other scenarios such as peers entering the network and lost messages.

4. NETWORK TRACK INITIATION USING TWO-PHASE COMMIT

Now that we have established typical properties of database systems (Section 2) and a method — two-phase commit — for extending those properties to distributed database systems (Section 3), we are poised to apply these database concepts to the network-centric tracking problem.

4.1 Track initiation versus maintenance

A simple but naïve implementation would add every track and associated measurement to a single distributed database, resulting in an exactly replicated track picture on every platform. While such an implementation would be conveniently accurate, it would also be prohibitively slow. Since every transaction imposed on the database must be approved by every peer, and the databases must be locked during the approval process, each transaction will take on the order of twice the average network latency (note the locked database segments for each platform in Figure 2b). With this in mind, if the whole database is locked during each transaction, the maximum transaction throughput (transactions per time) of the system is approximately $1/(2d)$, where d is the average network delay. In many practical tracking systems, this will preclude the possibility of committing every measurement to the database. For example a system with 10 sensors, each operating at a scan rate of 5 Hz, will generate 50 associated measurements per second per target. If the sensors are spread out and the accompanying network has a latency of half a second, then the maximum throughput will be one transaction per second, making the network 50 times too slow to handle all the observations for even a single target.

In light of the implausibility of simply adding every item of data to the distributed database, we now appeal to the previously established separation between network track *initiation* and *maintenance* algorithms. We will examine both the potential benefit and difficulty of handling each of the two classes of algorithms by using a distributed database methods.

Track maintenance in a network-centric environment is not without issues. Lost or delayed assigned measurement messages can lead to track state inconsistencies between peers on the network, though we take the view that the while loss or inconsistency of track maintenance messages leads to degradation in SIAP, the degradation at least grows gracefully with the number of lost messages. In addition, as previously observed, it is prohibitively costly to add *all* maintenance messages (assigned observations) to a distributed database, because the transaction throughput is more than the database can handle. Accordingly, we do not propose to impose the ACID constraints on track maintenance messages; instead we allow track maintenance messages to be carried via a *best effort* delivery, rather than the guaranteed synchronization offered by two-phase commit.

Track initiation, however, presents a different scenario. Initiation inconsistencies can lead to significant problems, such as dual tracks, unobserved targets, and inconsistently labeled tracks. These inconsistencies cause an immediate loss of SIAP and may result from even a single delayed message. For example two platforms may simultaneously and independently initiate a new track on the same target. If their initiation messages are delayed enough such that both send their messages before either receives the other's, a dual track will be created. Fortunately, initiation messages occur far less frequently than do maintenance messages, and consequently they may be handled by a distributed database.

Thus, the direction of our work has been on applying distributed database concepts to network track initiation only, leaving maintenance messages to be broadcast in an asynchronous manner without database locking.

4.2 Database replication

As mentioned in Section 3, databases are distributed in a variety of manners. For our purposes, namely achieving SIAP with network-centric trackers, we will assume the entire database of network tracks is replicated on each platform. This is for several reasons:

1. The database is small enough to easily be managed by a single computer.
2. Although a track's state may often change, once it is added to the database, the track's existence is only modified through track deletion. This allows us to make the optimization that database locks are not required when reading a track — an important optimization, as the number of database reads will vastly outnumber the number of database writes. A replicated database allows reads to be performed locally.
3. Each platform, having an entire copy of the database, is resilient to the failure of other platforms. If the database were fragmented, this would not be the case. There are, of course, various methods for maintaining the durability of the database, even in the event of node failure, but none of them allow a platform to function as autonomously as does replication.

4.3 The ACID properties for a network track database

We now briefly examine the distributed track initiation problem in light of each of the ACID requirements, assuming a completely replicated distributed database model.

Atomicity – This is a rather easy requirement to satisfy, since track insertions and updates are already dealt with by the tracker in an atomic manner.

Consistency – Several consistency constraints may be imposed on the distributed track database. For example, we impose the constraint that two different tracks with the same ID should never exist simultaneously in the database; this constraint is easy to satisfy by choosing track IDs to be a combination of the platform ID and a locally unique serial number.

More difficult to enforce is the constraint that the database contain at most one track for a given target. This constraint may only be satisfied insofar as any single local tracker can detect duplicate tracks. Numerica's multi-frame assignment tracker does an admirable, though not perfect in all cases, job of ensuring that local duplicate tracks are not initiated. The two-phase commit initiation scheme then allows local consistency to be transferred to the distributed system as a whole.

Isolation – The serial nature of a tracker's local operations ensures local isolation for that tracker's transactions, but isolation must still be ensured for the complete network of trackers. One way this may be accomplished is by locking the entire database of network tracks whenever a new track is added.

A more efficient method derives from standard database procedures. Traditionally, database applications only lock the rows (parts) of the database that concern a given transaction, allowing most other transactions to proceed, while blocking those that would interfere. We may take inspiration from this idea and lock *regions of space* rather than the entire database. That is, when a new track is initiated, we may lock the region surrounding that new track (say, a large sphere), suspending any other initiations within that region, but allowing new tracks

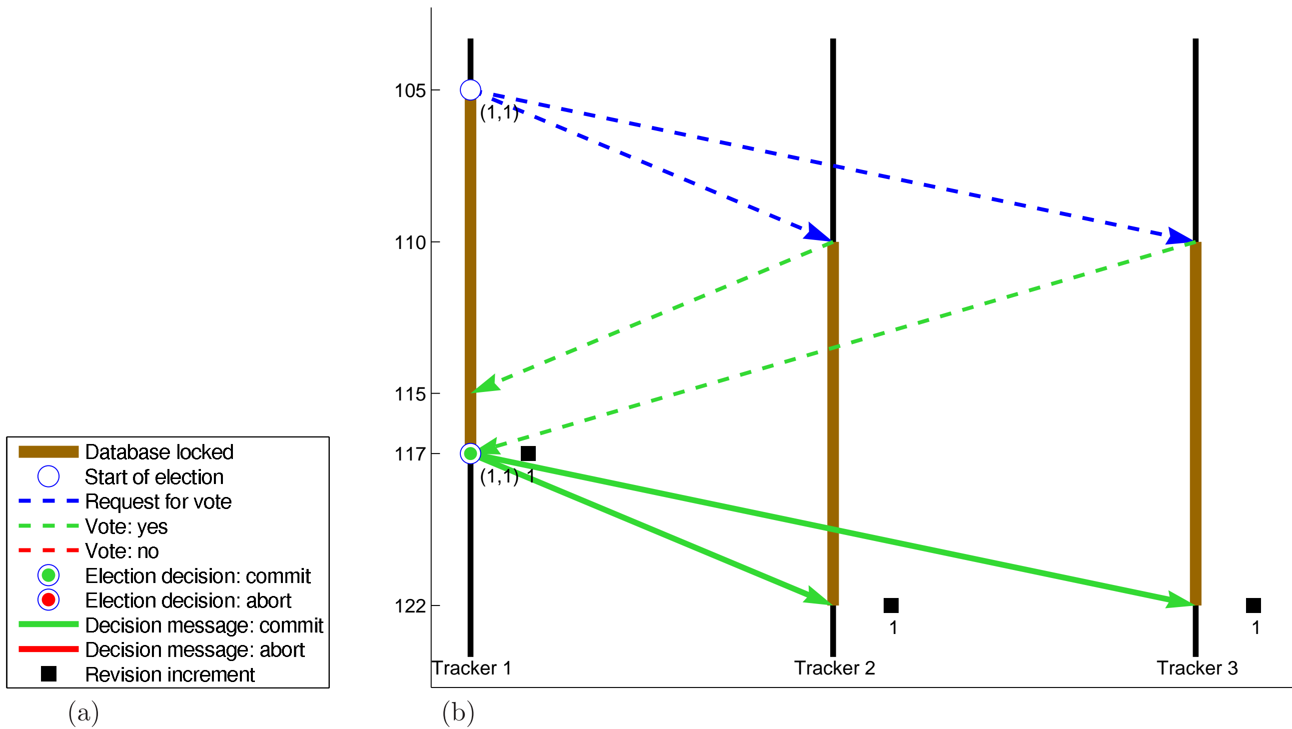
to be simultaneously initiated in other regions. Local region locking may greatly decrease the amount of time necessary for the initial start up of a wide-area system.

Durability – Ensuring durability requires each peer to agree on whether or not a transaction was committed or aborted; this is accomplished via the two-phase commit algorithm.

5. VISUAL EXAMPLES OF TWO-PHASE COMMIT INITIATION

Numerica has recently implemented the necessary infrastructure for multiple trackers to cooperatively initiate network tracks using the two-phase commit protocol discussed in this paper; here we present two examples of the algorithm in action, examining the messages that are passed between trackers and the internal state changes of those trackers. To more easily understand the progression of events, we will examine the visualizations in Figure 2 and Figure 3.

5.1 A Simple Example



(a) (b)

Figure 2. **(a)** Legend for interpreting the visualizations in (b) and Figure 3. **(b)** A simple example of the two-phase commit protocol executed amongst three trackers. In this example the transaction proceeds unhindered (i.e., there are no important race conditions resolved). See Section 5.1 for a discussion of this example.

We start with a simple example of the progress of the two-phase commit algorithm between three trackers, shown in Figure 2b. In this figure, the three trackers are represented by three vertical lines, and the time is shown along the left hand side, increasing towards the bottom of the figure. This example is provided as an illustration of the simplest possible case of a track being initiated across a network of three trackers. The events proceed as follows:

105s Tracker 1 decides to attempt a network track initiation; in order to do so, it starts the *(1,1)* transaction. In our nomenclature the first 1 indicates that the transaction was initiated by Tracker 1, and the second 1 reflects that it is the first transaction begun by that tracker. Tracker 1 locks its network track database, and sends a *Request for Vote* message to Tracker 2 and Tracker 3.

110s Trackers 2 and 3 receive the Request for Vote message, lock their network track databases, and send a *yes vote* back to Tracker 1.

115s Tracker 1 receives the yes vote from Tracker 2. At this point it cannot yet decide to complete the transaction, as it still has not heard from Tracker 3

117s Tracker 1 receives the yes vote from Tracker 3. As Tracker 1 now has yes votes from all of its peers, it resolves the transaction as a success. It commits the new network track as revision 1 in its network track database, after which its database is unlocked. Tracker 1 also transmits the *commit* message to Trackers 2 and 3 to inform them of the outcome of the transaction.

122s Trackers 2 and 3 receive the commit message from Tracker 1 and commit the new tracks to their databases, incrementing their revision to 1. The transaction is resolved, so they unlock their databases.

5.2 A More Complex Race Condition Example

Having looked at a simple example of the two-phase commit algorithm in action, we now turn our attention to a more complicated example. This example will showcase the algorithm's ability to correctly resolve a race condition between two trackers trying to initiate a network track, a race that in other architectures often produces redundant network tracks.

The scenario is pictured in Figure 3 and is from an actual simulation run. Figure 2a provides a legend for interpretation of the symbols and colors used in the diagram. In this example Tracker 1 and Tracker 3 both attempt to initiate a new network track. Tracker 3 succeeds, and Tracker 1 fails. We step through this example in time order below:

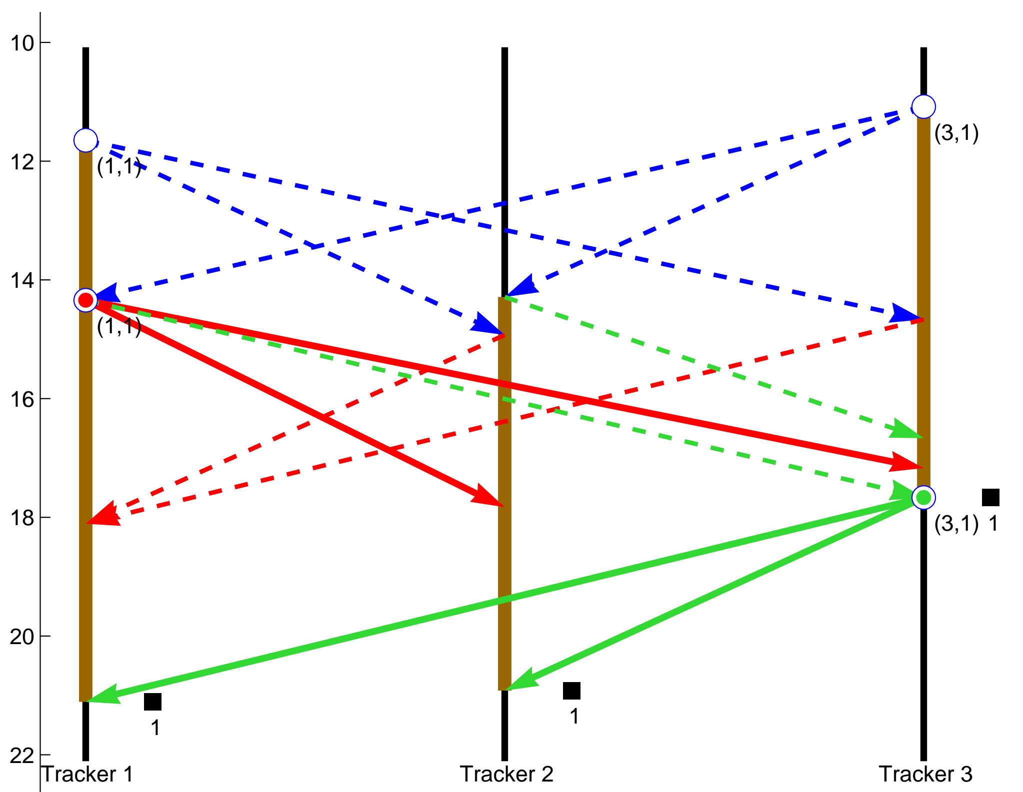


Figure 3. A more complicated two-phase commit example in which the algorithm successfully resolves a race-condition between Tracker 1 and Tracker 3. See Section 5.2 for a detailed description of this example.

11.1s Tracker 3 decides to initiate a network track. It starts transaction (3,1) and sends Request for Vote messages to Trackers 2 and 3. Because Tracker 3 was the first to begin its transaction, it is the tracker that will eventually get to initiate a new network track.

11.7s Tracker 1, unaware of Tracker 3's transaction in progress, decides to initiate its own network track, so it begins transaction (1,1) and sends Request for Vote messages to Trackers 2 and 3.

14.3s Tracker 2 receives the Request for Vote from Tracker 3 and responds by locking its database and sending a Vote: Yes to Tracker 3.

14.3s Tracker 1 receives the Request for Vote from Tracker 3 and, upon examination, determines that transaction (3,1) precedes its own transaction (1,1). Because (3,1) precedes its own transaction, it sends a Vote: Yes to Tracker 3 and aborts transaction (1,1), sending the abort message to Trackers 2 and 3.

14.7s Tracker 3 receives the Request for Vote from Tracker 1 and sends back a Vote: No because its own transaction supersedes transaction (1,1).

14.9s Tracker 2 receives Tracker 1's Request for Vote and responds with a Vote: No message, because it has already heard of transaction (3,1) and realizes this transaction supersedes transaction (1,1).

16.7s Tracker 3 receives Tracker 2's Vote: Yes and makes note of it. Tracker 3 cannot yet resolve transaction (3,1) because it must wait for Tracker 1's vote.

17.1s Tracker 3 receives Tracker 1's Abort (1,1) message and essentially ignores it.

17.7s Tracker 3 receives Tracker 1's Vote: Yes and, because it has received affirmative votes from all its peers, resolves its transaction (3,1) as successful. Tracker 3 sends out the Commit message to Trackers 1 and 2, increments its database to revision 1, and unlocks.

17.8s Tracker 2 receives Tracker 1's Abort (1,1) message and essentially ignores it.

18.1s Tracker 1 receives Vote: No messages from Trackers 2 and 3, which it may ignore because its transaction has already been aborted.

20.9s Tracker 2 receives Tracker 3's Commit message and commits the new network track to its database, which is now at revision 1. As the transaction has concluded, it unlocks its database.

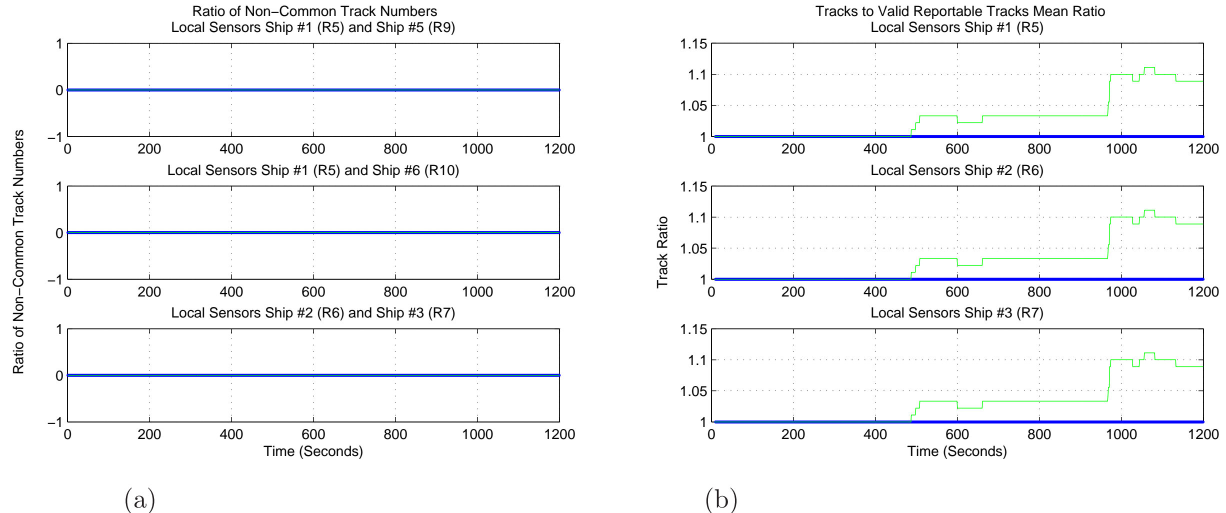
21.1s Tracker 1 receives the Commit and does the same, incrementing and unlocking its database. The three trackers now once again have consistent, matching network track databases.

6. NETWORK ARCHITECTURE REDESIGN SIMULATION RESULTS

To experimentally validate the algorithms presented in this paper, we performed simulations in a high fidelity air tracking benchmark. The scenario employed six radar sensors, each on a different ship, to track nine air targets over the course of 1200 seconds. In this section we will look at two metrics of particular import to network-centric tracking: the non-common track number ratio and the redundant track ratio. Both metrics were computed at approximately one second intervals throughout the 1200 second simulation run.

The non-common track number ratio is the fraction of tracks whose track numbers do not match between a given pair of sensor platforms. An important benefit of using the two-phase commit protocol (Section 3) is that the track numbers should match exactly by construction, and indeed, Figure 4a shows this to be the case.

The redundant track ratio, shown in Figure 4b, expresses the number of tracks being maintained by a given platform as a ratio to the number of valid tracks on that platform. For example, a tracker with five tracks for four targets would have a redundant track ratio of 1.2. Redundant tracks can arise out of a variety of situations; one situation of interest is when multiple networked trackers nearly simultaneously initiate a network track. In some architectures this race condition may lead to redundant network tracks. Redundant tracks can also arise due to purely local circumstances. Numerica's two-phase commit implementation prevents redundant network tracks due to simultaneous network initiation, and, while one can construct scenarios in which redundant tracks still occur due to local circumstances (such as redundant tracks from the local sensor), it does ensure that all platforms agree upon the redundant tracks. In Figure 4b a single monte carlo run is shown in blue; in this case no redundant tracks were created. The average over 10 monte carlo runs is shown in green. In this case some of the monte carlo runs did produce redundant tracks, however, all platforms agree upon the redundant tracks.



(a) (b)

Figure 4. Simulation metrics. For the sake of space, not all sensors are shown. **(a)** Non-common track number ratio metric. Not all pairs are pictured, but all pairs had the same performance. The two-phase commit algorithm, by construction, ensures the optimal performance of this metric; this is verified by noting that the blue lines (one run) and the green lines (average over 10 monte carlo runs) are both at zero, meaning that at no point did any tracker have a label for a track that was inconsistent with any other tracker. **(b)** Redundant track ratio metric. Note that, unlike in (a), here the two-phase commit algorithm does not guarantee perfect results for this metric. As discussed in Section 4.3, it only guarantees that redundant tracks will not be created by network race conditions. The blue lines show the redundant track ratio for a single monte carlo run and the green lines show the redundant track ratio averaged over 10 monte carlo runs. Some redundant tracks are produced in the 10 monte carlo run case, although all six platforms agree upon these redundant tracks.

7. CONCLUSION

Leveraging the vast DDS literature provides advantages to the network-centric tracking problem. For example, the two-phase commit protocol appears to be an effective replacement for the current network track initiation processes. Through the two-phase commit algorithm, we obtain provable resolution of a well defined class of race-conditions inherent in the communication of new system tracks to peers on the network. This resolution comes at the cost of a greater number of message exchanges, though the total bit count of the messages is approximately the same as many current algorithms, if not smaller. Furthermore, while additional rounds of messages increase latency, initiating tracks is already a process with relatively high latency. Several measurements need to be generated and processed, and the tracking architecture already holds newly initiated tracks for a span of time to confirm that they are firm before they are transmitted to the network.

Numerica has implemented and tested in simulation a version of the two-phase commit algorithm within its multi-frame assignment tracker, with favorable results.

ACKNOWLEDGMENTS

The authors acknowledge the SIAP JPO SBIR contract W9113M-06-C-0196. We also thank Aubrey Poore and Sean Roberts for their guidance and support of this work.

REFERENCES

[1] Paffenroth, R., Novoselov, R., Danford, S., Teixeira, M., Chan, S., and Poore, A., "Mitigation of biases using the schmidt-kalman filter," in [*Proceedings of SPIE Signal and Data Processing of Small Targets*], (2007).

[2] Bar-Shalom, Y., Chen, H., and Mallick, M., "One-step solution for the multistep out-of-sequence-measurement problem in tracking," *IEEE Transactions on Aerospace and Electronic Systems* **40**, 27–37 (January 2004).

[3] Chan, S. and Paffenroth, R., "Out-of-sequence measurement updates for multi-hypothesis tracking algorithms," in [*Proceedings of SPIE Signal and Data Processing of Small Targets*], (2008). Submitted for publication.

[4] Drummond, O. E., "A hybrid fusion algorithm architecture and tracklets," *Proceedings of SPIE* **3136**, 485–502 (1997).

[5] Drummond, O. E., Blair, W. D., Brown, G. C., Ogle, T. L., Bar-Shalom, Y., Cooperman, R. L., and Barker, W. H., "Performance assessment and comparison of various tracklet methods for maneuvering targets," in [*Proceedings of SPIE*], **5096**, 1–26 (2003).

[6] Özsu, M. T., [*Encyclopedia of Information Systems*], ch. Distributed Database Systems, 673–682. Academic Press (2003).

[7] Özsu, M. T. and Valduriez, P., [*Readings in Distributed Computing Systems*], ch. Distributed Data Management: Unsolved Problems and New Issues, 512–544. IEEE/CS Press (1994).

[8] Özsu, M. T. and Valduriez, P., [*Principles of Distributed Database Systems*], Prentice-Hall Inc. (1999).

[9] "Database." Web page: http://en.wikipedia.org/wiki/Database.

[10] "Two-phase commit protocol." Web page: http://en.wikipedia.org/wiki/Two_phase_commit.

[11] Böttcher, S., Gruenwald, L., and Obermeier, S., "Reducing sub-transaction abort and blocking time within atomic commit protocols," *proceedings of The 23rd Annual British National Conference on Databases* (2006).

[12] Gray, J. and Lamport, L., "Consensus on transaction commit," *ACM Transactions of Database Systems* **31**, 133–160 (March 2006).