A DISTRIBUTED BLOCKCHAIN LEDGER FOR SUPPLY CHAIN

A Thesis

Submitted to the Faculty

of

Purdue University

by

Haoyan Wu

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electric and Computer Engineering

August 2017

Purdue University

Indianapolis, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF THESIS APPROVAL

Dr. Zina Ben Miled

Department of Electrical and Computer Engineering

Dr. Brian King

Department of Electrical and Computer Engineering

Dr. Dongsoo Kim

Department of Electrical and Computer Engineering

Approved by:

Dr. Brian King

Head of Departmental Graduate Program

This thesis is dedicated to my parents, Baoquan Wu and Lijuan Kong, who have continuously supported and encouraged me when I was facing the challenges of graduate school and life. Without their unconditional love and wise guidance, it would have been hard for me to accomplish this work.

ACKNOWLEDGMENTS

I would like to thank my graduate advisor, Dr. Zina Ben Miled, for her guidance. Feedbacks from Dr. Brian King and Dr. Dongsoo Kim have been valuable in improving this research work. Special thanks to Mr. Jeffrey Tazelaar and Mr. John Wassick for their advice and clarifications on the relevant research issues. I also would like to extend my thanks to my friend, Jerry Li for his support.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure Page

LIST OF ALGORITHMS

Algorithm Page

ABSTRACT

Wu, Haoyan. MSECE., Purdue University, August 2017. A Distributed Blockchain Ledger for Supply Chain. Major Professor: Zina Ben Miled.

Affordable and reliable supply chain visibility is becoming increasingly important as the complexity of the network underlying supply chains is becoming orders of magnitudes higher compared to a decade ago. Moreover, this increase in complexity is starting to reflect on the cost of goods and their availability to the consumers. Optimizing the physical distribution phase in supply chain by providing increased visibility to trading partners can directly reduce product cost. Current supply chain information systems often lack the ability to cost-effectively relay ground truth information in near real time to all stakeholders and most importantly to the supplier and the customer during the transport of the shipment. This thesis presents a solution that addresses this gap through a distributed architecture. The solution enables small, medium and large businesses to interact in a dynamic and shipment-centric manner through a private blockchain sub-ledger that digitizes the transfer of custody for each shipment. Information in this private ledger is augmented by a public event ledger that reflects the movement of the shipment in real time. Third party monitors are engaged in the validation of the geolocation of the shipments by posting their physical proximity in the form of events to the public ledger.

1. INTRODUCTION

1.1 Motivation

Supply chains span many geographies, modes and industries and involve several phases where data flows in both directions from suppliers, manufacturers, distributors, retailers, to customers. This data flow is necessary to support critical business decisions that may impact product cost and market share. Current centralized supply chain information systems are unable to support the needed real-time transparency, scalability and security. This thesis proposes a distributed private ledger solution for the sharing of information among trading partners in pseudo real-time. The proposed system uses a distributed ledger which is based on blockchain to document and exchange custody events related to shipments. These events are only shared among trading partners, thereby protecting the privacy of the participants. A private ledger is created for each shipment allowing the overall network to support an increasing number of shipments through a hybrid peer-to-peer communication model.

1.2 Background

Supply chain consists of several phases involving several companies. These phases include planing, development, manufacturing, delivery and returns [1]. Planing is the first phase of the supply chain process and focuses on meeting customer demands. The second phase is concerned with a development plan for the product including marketing and pricing. The manufacturing phase includes design, production, testing and packaging. The first two steps are usually handled by an enterprise resource planning system (ERP) [2]. The third phase is handled by several product life cycle management information systems (PLMS) [3]. The fourth phase focuses on the

delivery of the products. It starts with a pre-shipment step which consists of the preparation of product for shipment by the supplier followed by a transportation step which is primarily managed by a carrier.

There are also three main flows in supply chain: material flow, information flow and capital flow. Material flow is concerned with the transfer of the physical product from the supplier to the customer via a carrier. Information flow is the digital thread that documents the transfer of the product. This digital thread includes the order information and the shipment information (e.g., shipment status, GPS location, etc.). Capital flow is concerned with payment and transfer of the ownership of the asset.

Supply chain visibility refers to the accurate and real-time delivery of information to all stakeholders. The benefits of an increasing supply chain visibility include reduced business and supply chain risks, improved lead times and performance, and early identification of shortage and product quality problems [4]. Lack of visibility and transparency can cause inaccurate forecasting and unexpected delays leading to increased product cost.

1.3 Approach

This thesis focuses on developing a distributed and scalable data model for the sharing of supply chain information during the transport phase of supply chain. This model is based on the blockchain technology which is implemented using a heterogenous set of NoSQL database management systems. The approach addresses the heterogeneity across the databases of the different nodes participating in the information flow by using the EDI-214 standard [5] translated into a JSON data representation [6]. EDI-214 is a standard for the electronic exchange of the transportation carrier shipment status.

1.4 Organization

Chapter two provides a summary of related work with a focus on blockchain, NoSQL and supply chain. Chapter three introduces the data model used in the proposed solution and describes how events are translated into records in the distributed ledger. Chapter four presents the implementation of the data model for X86 nodes by using MongoDB [7] and for mobile nodes by using Couchbase Lite. This chapter also discusses how interoperability among the different database management systems is achieved in the proposed system by using the EDI-214 standard. Moreover, the chapter includes examples of practical test scenarios of the proposed framework. Chapter five concludes with the main contributions of this work and proposes direction for future work.

2. RELATED WORK

In the past three decades, the digitization of information and its exchange went through several phases that were driven by demands and technological innovations. Going from paper-based records within an institution to isolated applications that cover one or more aspect of the business was an initial phase. During this phase, multiple applications and underlying databases were used for accounting, human resources, maintenance, etc. This phase was followed by a trend towards enterprise resource planning (ERP) systems that integrated the digital thread across various activities within each institution including accounting, human resources management, payroll, maintenance, warehousing and logistics with varying emphasis depending on the core business of the organization. ERPs facilitate information flow between all business functions within an organization and manage transactions with partner organizations [8]. Additional advances and demands led to the globalization of these ERP systems especially for multi-nationals and corporate groups. Moreover, the digital thread started to extend to cover the external relations of the organization with its customers (e.g., customer portals and customer relations management systems - CRMs) and its partners (e.g., suppliers, financial institutions, service providers). This latter extension was facilitated by the emergence of the cloud computing business model which allows different partners to exchange information using a common platform. These platforms are nowadays seamlessly integrated within the business processes internal to an organization. Compared to their predecessors, these new systems offer significant improvements in efficiency and transparency. However, with the recent advances and ubiquity of sensor technology, there are increasing demands for higher efficiency and transparency.

2.1 Supply Chain

E2Open [9] and SAP [10] are examples of leading edge supply chain systems that provide well-developed functionalities including pseudo real-time shipment information and interoperability with multiple enterprise resource planning (ERP) systems. These systems are cloud-based and adopt a centralized and often complex software and hardware architecture. Given the large amount of traffic and the variety of stakeholders (i.e., small, medium and large businesses) that need to be supported by a global supply chain system, a centralized architecture has limitations with respect to both scalability and affordability. As a result of these limitations, small and medium business have a difficulty penetrating new and well established markets.

Supply chain operating networks (SCONs) are platforms that enable multiple partners to exchange information. SCONs are cloud-based solutions and often seamlessly integrate with information systems local to each partner. They are commonly powered by solutions such as E2Open and SAP. Unfortunately, the value derived and the success of SCONs are directly proportional to the level of participation by a given industry sector or trading group. Furthermore, the competitiveness of a given company is directly related to its ability to venture into new industries and sectors which today requires the capability to support multiple SCONs, often leading to inefficiencies and high costs [11].

The main standard used in the exchange of information among trading partners is the EDI standard [12]. SCONs, for instance, use this standard. The EDI standard digitally captures the content of various documents used in the supply chain including invoices, purchase orders, bill of lading as well as messages such as the Carrier Shipment Status Message.

2.2 Bolckchain

Blockchain is a distributed database that maintains a continuously growing list of records, called blocks, secured from tampering and revision [13]. Blockchain was first

used to support the digital currency BitCoin [14]. It was later adopted by other digital currencies and was the subject of highly publicized successes and failures. At the core of the blockchain technology is a distributed ledger with two types of transactions. A single genesis transaction which creates value and a transfer transaction that transfers value from one party to another. Each transaction is digitally signed by the issuer and posted to the global ledger. A group of transactions are then collected into a block, the block is validated by a third party (a miner) and is locked. This mechanism represents the strength of the blockchain technology. Each block in the chain is immutable since it is linked to its predecessor and any change to any of the blocks invalidates all the blocks downstream in the chain. Furthermore, the more mature the block is (i.e., the longer it has been in the global ledger chain), the greater is its integrity. Each participant in the global network keeps a copy of the ledger and every time a new block is created, it is broadcasted to all the participants that add it to their local copy of the ledger.

In general, participation in the global ledger is anonymous as each party is identified by a digital ID. From a business perspective, issuers and beneficiaries are encouraged to participate in the ledger because of this anonymity, the lack of a central controlling party, reduced transaction fees and the real-time execution of the transactions. Miners are also incentivized because they receive a fraction of the transaction fee for every block they validate.

The blockchain technology can be implemented either on a distributed network architecture or a centralized cloud-based architecture. The distributed architecture often adopts a peer-to-peer network communication model where all nodes in the network have equivalent roles and privileges. Nodes or peers in the network cooperate to service each others request with a limited or no centralized management. Several well-known software applications use this architecture including Bitcoin, the file sharing applications bittorrent [15], Napster [16] and the messaging application Skype [17].

Blockfreight is an example of supply chain solution that implements blockchain on a distributed peer-to-peer network. It proposes an end-to-end container tracking solution. There are several differences between Blockfreight and the framework proposed in this thesis. First, Blockfreight is a solution for international trade (i.e., mostly sea shipping) while we focus on local trade (i.e., truckload shipping). This difference causes a significant divergence in the requirements of the two systems. For instance, transparency in international trade may be measured in days whereas for local trade it is measured in hours. Second, Blockfreight relies on an open distributed global ledger. Even though participation in the global ledger is anonymous, we believe that this participation raises privacy concerns with most businesses. Our proposed solution addresses privacy issues by combining a private shipment-centric custody sub-ledger with a movement tracking general ledger.

Blockchain solutions for supply chain have also been proposed over centralized networks (e.g., [18]). These solutions avoid the challenges associated with an open and distributed peer-to-peer architecture but also dismiss the intended resilience, scalability and affordability of the distributed network.

In general, current research around blockchain focuses on facilitating three main functionalities: tracking (e.g., provenance, proof of origin), transfer (e.g., smart contracts) and payment (e.g., digital currency). Recently, Ethereum [19], a peer-to-peer blockchain programming language has been proposed to support all three.

2.3 Database Management Systems

Relational databases dominated for a very long time and where the de-facto data model for most data management solutions. However, demands for solutions that can accommodate flexible schema with heterogeneous data started to increase. These demands lead to the emergence of NoSQL databases.

The main difference between NoSQL databases (e.g., MongoDB and Couchbase Lite [20]) and relational databases is in the way data is organized and stored. While

relational databases store data in tables consisting of columns of attributes and rows of records, NoSQL databases store data in documents. Each document contains one or more fields with a typed value such as string, data, array or even a nested document. A document is a record and represents the basic unit of data in a NoSQL database. A collection represents a group of documents and may be considered as the counterpart of a table in a relational database.

The schema of a database is the structure of the objects in the database and it is often derived from business requirements. For a relational database, a schema consists of a set of tables. For example, a simplified database for orders (Fig. 2.1) may consist of three tables: *Orders* which includes order information, *productDetails* which includes the information related to a product and *orderLinker* which connects the first two tables. Once designed and populated, the schema is rigid and cannot be easily modified. Adding or modifying the attributes of the schema requires a complex migration from the source schema to the target schema. This lack of flexibility places a significant burden on the database designer to make sure that the relational schema can accommodate current and potential future requirements.

NoSQL databases avoid the above limitation by foregoing the use of a schema. NoSQL databases are in fact also known as schemaless databases or databases with dynamic schema. There is no predefined relations in the NoSQL databases. For example, mongoDB provides a document data model that allows the combination of any type of data [21].

Fig. 2.1 and Fig. 2.2 compare the two types of databases with a specific example. The SQL database uses three tables to store order information. As shown in Fig. 2.1, the orders table has several attributes related to the basic information of orders, where *orderID* is the primary key. The *productDetail* table contains attributes related to the individual products where *productID* is the primary key. The table *orderLinker* links the *orderID* table and the *productDetails* table and uses two foreign keys. The third table is needed in order to meet the normalization requirement of database management systems since one order may have multiple products.

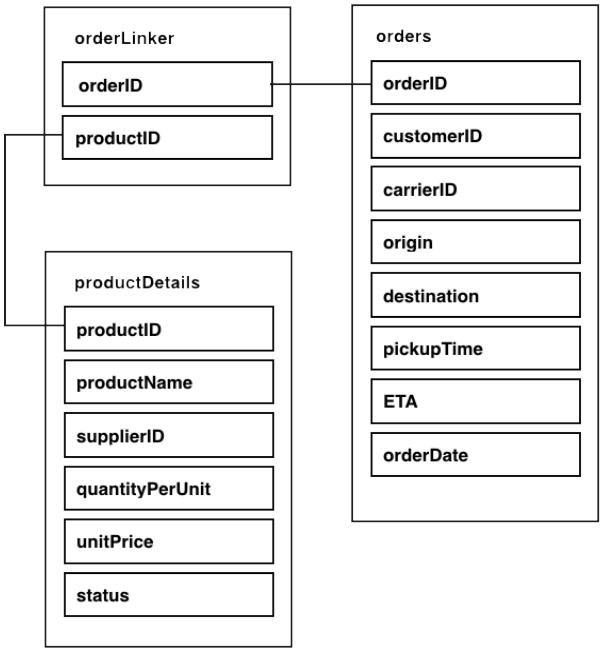

Fig. 2.1. SQL Example

The document-oriented NoSQL database uses one JSON formatted document for each order. Documents are grouped into collections. Collections are containers that hold related documents. Fig. 2.2 shows the structure of a document for a given order. Since NoSQL databases support embedded structure, an array in the document called product can hold multiple sub-documents related to the product information. With this structure, if a query for a product is issued, there is no need to perform a join. The join operation is however necessary in SQL databases. A single document can contain all information about an order in a NoSQL database. Furthermore, if new fields need to be added, they are simply inserted in the database. In contrast, adding a field in an SQL database requires the redesign of the schema and the migration of the data from the old to the new database. In summary, the benefits of a NoSQL database include the ability to handle unstructured data and to adapt to evolving business requirements.

```
{
   "orderlD": "57dafad59b8f13c51c80ad45",
   "cusotmerlD": "57dafad59b8f13c51c80ad46",
   "carrierlD": "57dafad59b8f13c51c80ad47",
   "supplierlD": "57dafad59b8f13c51c80ad48",
   "origin": "Indianapolis",
   "destination": "San Francisco",
   "pickupTime": "04/05/2017 16:40 PDT",
   "ETA": "04/07/2017 17:00 EDT",
   "orderDate": "04/01/2017 10:00 EDT",
   "product": [
       {
             "prodcutlD": "77cefad69b8333c51990rf11",
             "prodcutName": "water",
             "quantityPerUnit": "10",
             "unitPrice": "$10.34",
             "status": "liquid"
       },
       {
             "prodcutlD": "77cefad69b8333c51990rf12",
             "prodcutName": "butter",
             "quantityPerUnit": "12",
             "unitPrice": "$10.00",
             "status": "solid"
       }
       {...}
   ]
}
```

Fig. 2.2. NoSQL Example

3. SYSTEM MODEL

The proposed supply chain visibility framework is motivated by real industry demands and designed according to actual business requirements. The framework consists of a set of dynamic sub-ledgers and a central public ledger. The sub-ledgers are private to the trading partners and a sub-ledger is created for each shipment. Participation in the sub-ledger is based on the partners involved in the execution of the corresponding purchase order. The public ledger is open to all and contains tracking information for all trucks transporting shipments. Both the sub-ledgers and the public ledger are distributed and each participating node keeps a copy of the ledger of interest. This chapter describes the data model used for both type of ledgers and the mechanism used to keep the information in these ledgers up-to-date.

3.1 Physical Distribution Workflow

The focus of the proposed system is to provide real-time visibility during the physical distribution segment of the supply chain. Physical distribution is concerned with the transfer of goods or products from the supplier to the customer. The process underlying this transfer includes several steps and various stakeholders. In this thesis, we restrict the trading partners to only three stakeholders, namely, the supplier, the customer and the carrier. Typically, several other stakeholders are involved including more than one carrier as well as freight forwarders and freight brokers.

Each physical distribution transfer begins with a customer issuing a delivery note. Once the delivery note is received by the proposed system, a shipment centric peer-to-peer sub-network is dynamically established to include the customer, the carrier and the supplier. Subsequently, the order information is forwarded to the partners involved in the shipment. Once the shipment is picked up, events are generated to

update the status of the shipment and this information is shared with stakeholders until the shipment is delivered to its destination. For example, an on-board message is not only processed internally by the carrier but also sent to the supplier and the customer. The physical distribution process ends with the customer confirming the receipt of the goods.

3.2 Architecture Overview

The architecture of the proposed hybrid peer-to-peer physical distribution (HP3D) framework consists of a collection of dynamic sub-networks. These sub-networks are created on demand, emulate the end-to-end movement of the shipment and terminate when the delivery of goods is completed. Each sub-network allows stakeholders to share information related to a given shipment and provides them with pseudo real-time visibility in the physical distribution segment of the supply chain. There is also a global network that is open to all partners as well as third-party monitors. This global network is persistent and does not terminate. It is used as a timestamp and a proof of record for the events related to all shipments. There are four types of participants in HP3D: index server, peers, administrative nodes and external monitors.

The index server is a central directory that maintains the IP addresses for all the nodes in the network. It also assigns a unique ID to each participant. Peers are nodes that can take on different roles in different shipments (e.g., supplier, customer, or carrier). However, all the peers share a unified architecture regardless of their roles. Peer applications have a three-tier architecture consisting of:

- A presentation tier which allows the users to interact with the proposed system such as submit or review incoming events, communicate with the index server and register in the sub-network.

- A middle tier which handles and processes user requests and communicates with the other peers in the network.

- A data tier which stores and maintains the shipment-centric sub-ledgers.

A special node, called administrative node, is also needed in the HP3D. This node has the same three-tier architecture as the other nodes. Each trading partner has one administrative node which is responsible for communicating with the enterprise resource planning system (ERP). The administrative node participates in all sub-network involving the associated partner and maintains a permanent record of all information exchanges related to the partner's shipments.

Third party external monitors are the fourth type of participants in the proposed system. They are engaged in the validation of the geolocation of the shipments by posting their physical proximity in the form of events to the public ledger. The external monitors have an architecture similar to that of the peers with the exception of the presentation tier since the monitors do not need to query shipment data.

3.3 Events

The proposed framework supports three types of events:

- Genesis event: This event indicates the start of a shipment. It is initiated by the administrative node, transformed into JSON format and broadcasted to all trading partners. Each partner that is participating in the shipment, stores this information into its local database in the form of a document. Since the administrative node communicates with the ERP which is responsible for the initiation of an order, the parser in the administrative node translates the information from the ERP format to an EDI-214 standardized JSON formatted message. The details of the genesis event are only accessible to the trading partners. However, a hash value calculated from the genesis event is posted to the public ledger.

- Custody event: This event records the custody status of the current shipment. The custody can remain with the current holder of the shipment or transfer it

from one participant to another. (e.g., shipment transferred from supplier to carrier, shipment delivered to customer by carrier). In addition to the genesis event, the custody events form the shipment-centric private ledger which is shared among the supplier, carrier and customer for a given shipment. Similar to the genesis event, the hash value of each custody event is calculated and posted to the public ledger.

- Monitoring event: Monitoring events indicate the geographical status of a shipment. This information is generated by external monitors when trucks are physically near the monitors and an information exchange is executed between the monitors and the trucks in order to record this physical proximity. The monitoring events are posted to the public ledger by the external monitors.

3.4 Advanced Data Structure

HP3D uses three main data structures each identifying a different type of nodes in the network, namely: node information, participant information and monitor information.

The node information data structure is used by the peers in the network to request services from the index server. This data structure includes seven fields as shown below (Fig. 3.1):

```
type NodeInfo struct{
    ReqType int
    Timestamp time
    IPAddress string
    Key []byte
    IMEI []byte
    PublicKey []byte
    AffiliatedCom string
    NodeID bson.ObjectId `bson: ``_id,omitempty'' `
}
```

Fig. 3.1. Node Information Data Structure

- *Reqtype* is an integer value used to denote the type of a request. It is checked by the index server when a message is received. *Reqtype* can take on three integer values: a) 0 for an incoming message which is used to update the IP address of a node, b) 1 for an IP address query request for another node and c) 2 for mobile node verification based on an *IMEI* number.
- The *Timestamp* field represents the time that the IP address of a given node is updated.
- The *IPAddress* field is a string that records the IP address of a node.
- The *Key* and *IMEI* fields are byte arrays which are used in the mobile verification routine.
- The *PublicKey* stores the public key of a node used for verifying events signature.
- *AffiliatedCom* is used to identify the company the node is affiliated with.
- The *NodeID* field contains a node ID uniquely assigned by the index server.

The *OrderPart* information (Fig. 3.2) is used to support the second type of messages. The *OrderPart* structure uses *OrderID* as a unique ID and also contains a list of all the companies associated with the order. This data structure is used in the creation and maintenance of the dynamic sub-network associated with each shipment.

```
type OrderPart struct{
    OrderID bson.ObjectId `bson:"_id,omitempty"`
    Part []Participants
}
type Participants struct{
    CompanyName string
    CompanyID bson.ObjectId `bson:"_id,omitempty"`
}
```

Fig. 3.2. Participants Information Data Structure

The third kind of data structure is related to the monitors. *MonitorInfo* has three fields as shown in Fig. 3.3. *MonitorID* is the unique ID of the external monitor, *MonitorIP* records the IP address of a monitor and *MonitorLocation* is a sub-structure that contains the geological information of the monitor. This geolocation information is also the one that will be reported as the shipment location when the truck carrying the shipment is physically near the monitor.

```
type MonitorInfo struct{
    MonitorID bson.ObjectId `bson:"_id,omitempty"`
    MonitorIP string
    MonitorLocation GPS
}
type GPS struct{
    GPSCordsX string
    GPSCordsY string
}
```

Fig. 3.3. Monitors Information Data Structure

HP3D also has three kinds of data structure for exchanging and storing events and blocks. These are *PrivateGCEvents*, *PublicEvents* and *Blocks*.

PrivateGCEvents is the structure used for the genesis and custody events messages which are exchanged across trading partners and is used to update the copy of each private ledger in the participating nodes. This data structure (Fig. 3.4) has three fields, a unique ID, a timestamp for the event and a sub-structure containing the details of either a genesis event or custody event. This sub-structure is based on the EDI-214 standard.

```
type PrivateGCEvents struct{
    EventID bson.ObjectId `bson:"_id,omitempty"`
    Timestamp string
    EDI214 InterchangeEnvelope
}
```

Fig. 3.4. Private Genesis and Custody Event Data Structure

PublicEvents (Fig. 3.5) is the structure used in the public ledger to represent the three events in the proposed system. For the genesis events and the custody events, *EventHash* is the field that records the hash value of the event. This hash value is derived from the message in the *EDI214* format which is discussed in Section 3.5. The trading partners could check the validity of these two types of the events by comparing the hash value in the public ledger with the one computed from the *PrivateGCEvents* data structure in the private ledger.

```
type PublicEvents struct{
    EventID bson.ObjectId `bson:"_id,omitempty"`
    Timestamp string
    MonitorData
    EventHash [32]byte
    PreEveHash [32]byte
    CurEveHash [32]byte
}
type MonitorData struct{
    MonitorID bson.ObjectId `bson:"_id,omitempty"`
    TruckID bson.ObjectId `bson:"_id,omitempty"`
    GeoInfo GPS
}
```

Fig. 3.5. Public Events Data Structure

```
type Blocks struct{
    BlockID bson.ObjectId `bson:"_id,omitempty"`
    Event []PublicEvents
    Timestamp string
    Nonce int
    PreBloHash [32]byte
    CurBloHash [32]byte
}
```

Fig. 3.6. Blocks Data Structure

Monitoring events use *MonitorData* and *EventHash*. *MonitorData* (Fig. 3.5) is a sub-structure that contains the information of the monitor, the truck and the

geolocation. *EventHash* is computed from the *MonitorData*. The fields *PreEveHash* and *CurEveHash* are hash values for chaining the events and building the blocks in the public ledger. These two values are calculated locally by each node.

Events in the public ledger are grouped into block using a data structure called *Blocks* (Fig. 3.6). *BlockID* is the identifier of the block. Since a block may contain multiple chained events, the *Event* field is an array of *PublicEvents*. The field *Timestamp* corresponds to the time of the block. The *Nonce* is an arbitrary integer value computed for each block. It increases the complexity of computing the hash value. For instance, the nonce can impose that *CurBloHash* includes a number of leading zeros. *PreBloHash* has the hash value of *CurBloHash* from the previous block.

3.5 EDI-214 Standard

The proposed system uses the JSON format when transferring messages. These messages adhere to the EDI-214 standard. However, the EDI-214 standard message is traditionally represented as a string. Therefore, and in order to increase the compatibility of the proposed system with legacy systems, a parser is used to translate this message from a string representation to a JSON representation and vice-a-versa.

The hierarchical structure of EDI-214 message [22] has three envelopes. An envelope starts with a header and ends with a trailer. The shipment status information is encapsulated inside the *Transaction Envelope*.

The proposed system uses JSON message for communication and JSON-like format BSON message for storing information into the database. Therefore, the system uses a JSON representation of the EDI-214 message. The following data structures are designed to accommodate EDI-214 messages.

3.5.1 Interchange Envelope of EDI-214

The EDI-214 message has three envelopes. The outermost envelope, labeled *Interchange Envelope* (Fig 3.7), has three fields: *Header* (Fig. 3.8), *Envelope* and *Trailer* (Fig. 3.8) which are transferred to the JSON data structure.

```
type InterchangeEnvelope struct{
    Header InterchangeHeader
    Envelope []GroupEnvelope
    Trailer InterchangeTrailer
}
```

Fig. 3.7. Interchange Envelope of EDI214 Data Structure

```
type InterchangeHeader struct{
    ISA ISAattr
}
type InterchangeTrailer struct{
    IEA IEAattr
}
```

Fig. 3.8. Header and Trailer of Interchange Envelope

The interchange envelope acts like a container that holds the functional group discussed in Section 3.5.2. One or more functional group can be included in the envelope. The *Header* and the *Trailer* have the *ISA* and the *IEA* sub-structures. Attributes are stored as strings inside the two sub-structures and indicate the message sender and receiver as well as the message version.

3.5.2 Functional Group Envelope

The functional group consists of similar transaction sets. This group (Fig. 3.9) is defined by its *Header* and *Trailer* (Fig. 3.10). The transaction sets are stored inside the *Envelope*.

```
type GroupEnvelope struct{
    Header FunGroupHeader
    Envelope []TransEnvelope
    Trailer FunGroupTrailer
}
```

Fig. 3.9. Functional Group Envelope Data Structure

For example, the header of a functional group has the eight attributes shown in Fig 3.11. *GS01* an ID code indicates the type of transaction of the current functional group (e.g., PO means Purchase Order). *GS02* and *GS03* identify the sender and the recipient. *GS04* and *GS05* correspond to the date and time. *GS06* refers to the functional group control number which is same as the one in this functional group's trailer (*GE* in *FuncGroupTrailer*). *GS07* indicates the responsible agency. *GS08* has the version or release identifier code for the standard.

```
type FunGroupHeader struct{
    GS GSAattr
}
type FunGroupTrailer struct{
    GE GEattr
}
```

Fig. 3.10. Header and Trailer of Functional Group Envelope

```
type GSatt struct{
    GS01 string
    GS02 string
    GS03 string
    GS04 string
    GS05 string
    GS06 string
    GS07 string
    GS08 string
}
```

Fig. 3.11. Attributes of Functional Group Header

3.5.3 Transaction Envelope

The *TransEnvelope* (Fig 3.12) has the similar structure as the previous two envelopes including *Header*, *Envelope* and *Trailer*.

```
type TransEnvelope struct{
    Header TransHeader
    Envelope Loops
    Trailer TransTrailer
}
```

Fig. 3.12. Transaction Envelope Data Structure

The *TransHeader* structure has three fields. The *ST* in the *TransHeader* data structure in Fig. 3.13 is the transaction set header indicating the beginning of a transaction set. *B10* refers to the beginning segment for the transportation carrier shipment status message which is represented by the *Loops* sub-structures. *MS3* refers to interline information and is optional. *SE* in the *TransTrailer* indicates the end of the transaction set.

```
type TransHeader struct{
    ST STAattr
    B10 B10attr
    MS3 []MS3attr
}
type TransTrailer struct{
    SE SEattr
}
```

Fig. 3.13. Header and Trailer of Transaction Envelope

There are three fields in *B10attr* (Fig. 3.14). *B1001* is the invoice number assigned by the carrier, *B1002* is the shipment identification number that is uniquely assigned by the supplier and *B1003* is a code that uniquely identifies a carrier.

The shipment status as previously mentioned is stored in the *Loops* sub-structure which are discussed in the next section.

```
type B10att struct{
    B1001 string
    B1002 string
    B1003 string
}
```

Fig. 3.14. Attributes of B10

3.5.4 Shipment Status

There are four loops used to represent the details the shipment status in EDI-214. These are *Loop1000*, *Loop1100*, *Loop1200* and *Loop1300* shown in Fig. 3.15.

```
type Loops struct{
    L1000 []Loop1000
    L1100 []Loop1100
    L1200 []Loop1200
    L1300 []Loop1300
}
```

Fig. 3.15. Loops for Shipment Status

The fields in *Loop1000* include transaction instructions such as the bill of lading, handling instructions or unexpected lading.

The fields in *Loop1100* (Fig. 3.16) specify the status of a shipment, such as the reason for a status, the date and the time of the status and the date and time of scheduled appointments. *AT7* in *Loop1100* indicates the shipment status details including the status, the reason for the status and the time the status was reported. *MS1* refers to the location of the shipment consisting of either a) city and state, b) longitude and latitude, c) postal code at the time the shipment status message was sent. *MS2* refers to the equipment or container owner and type. Field *M7* field contains the seal number used to close a shipment and is given by a supplier to a carrier. If the seal is broken and the shipment is open (e.g., by customs), a new seal number is assigned.

```
type Loop1100 struct{
    AT7 AT7attr
    MS1 MS1attr
    MS2 MS2attr
    M7 M7attr
}
```

Fig. 3.16. Loop1100 Data Structure

Loop1200 (Fig. 3.17) is used to identify a participant in terms of the company's name (*N1*, *N2*), address (*N3*) and geographic location information (*N4*).

```
type Loop1200 struct{
    N1 N1attr
    N2 N2attr
    N3 N3attr
    N4 N4attr
}
```

Fig. 3.17. Loop1200 Data Structure

```
type N1attr struct{
    N101 string
    N102 string
    N103 string
    N104 string
}
```

Fig. 3.18. Attribute of N1

N1 consists of multiple fields as shown in Fig. 3.18. *N101* is an identification code consisting of four components: *CN* (Consignee), *SF* (ship from), *SH* (shipper) and *ST* (ship to). *N102* contains the name of the company or the individual associated with*N101*. *N103* and *N104* are qualifier and identification codes.

Loop1300 indicates the order identification details. It only has one field called *OID* (Fig. 3.19) which specifies the purchase order number (*OID02*), lading quantity (*OID05*), weight unit (*OID06*), weight (*OID07*) and volume (*OID09*). Other fields are associated codes or qualifiers.

```
type OIDattr struct{
    OID01 string
    OID02 string
    OID03 string
    OID04 string
    OID05 string
    OID06 string
    OID07 string
    OID08 string
    OID09 string
}
```

Fig. 3.19. Attributes of OID

3.6 Blockchain

Traditionally, supply chain partners can do business by exchanging documents. This is actually the norm today in various countries and often for most international trades. This process is inefficient, slow, and can cause delays and increase product cost. In this thesis, we propose to use the blockchain technology, in order to make the above data flow process efficient and resilient. A blockchain ledger can be considered as a distributed database that holds a continuously increasing list of events grouped into blocks [23]. The proposed framework has two types of ledgers: a private ledger and a public ledger. The private ledger contains the details of the genesis events and the custody events which are shared among trading partners. These events are not chained thereby limiting the computing requirements for nodes belonging to trading partners. The validity of each of these events can be easily verified by comparing the hash value of the event in the private ledger to the corresponding hash value in the public ledger. The public ledger acts as a reference and proof of validity for the genesis events and the custody events in addition to being a permanent record for monitoring events.

The local database of each node has five collections: *NodeInfo*, *OrderParticipants*, *PrivateEvents*, *tempPublicEvent*, *Blocks* and *tempBlocks*.

- *NodeInfo* is used for storing the information about the node and allows the node to communicate with the index server and other nodes in the network.
- *OrderParticipants* records the participants information. It contains an order ID and a sub-structure consisting of all the partners involved in an order.
- *PrivateEvents* captures the details of a genesis event or a custody event.
- *tempPublicEvent* is used for temporarily storing events that are not yet part of a block. Before a block is built, recent events are chained and stored in *tempPublicEvent* until the block is completed.
- *Blocks* is used for chained blocks. The longest chain is accepted by every node in the network as the public ledger of record.
- *tempBlocks* is used when multiple blocks are received at same time, or the *PreBloHash*block received does not match the latest block in the local database.

The local database of each monitor has four collections: *MonitorInfo*, *OtherMonitors*, *tempPublicEvent* and *Blocks*.

- *MonitorInfo* is used to store the information of the monitor. The document uses the *MonitorInfo* structure shown in Fig. 3.3.
- *tempPublicEvent* is used for temporarily storing events that are not yet part of a block. Before a block is built, pending events are chained until a new block is completed.
- *Blocks* is used for storing chained blocks. It has the same structure as the one used in the nodes.
- *tempBlocks* has the same purpose and structure of *tempBlocks* in a node's database.

Basically, monitors are responsible for building the public ledger. Since they do not generate genesis events or custody events, the local database of monitors does not include the *PrivateEvents* collection.

There are two kinds of information validation in the proposed system. The first is the validation of events. The second is the validation of blocks. The events are checked every time they are received. Therefore, there is no need to check them again when they are chained or combined into a block in the public ledger.

The genesis event indicates a new shipment generated by an ERP. The administrative node is the only node that can generate a genesis event. Since this event initializes a new shipment and establishes the order ID, the customer, the supplier and the carrier, the genesis event does not need to be validated.

The custody event may involve two participants from two different organizations or companies. This event is signed with the participants private keys before it is sent. Normally, node A of the first participant signs the event first, then node B of the second participant signs the message that has been previously signed by node A. Node B then sends the doubly signed custody event and is considered as the event generator. The details of a custody event is broadcasted to all trading partners in a given shipment by the event generator. The nodes who receive the private custody event check the message by using node A and node B's public keys. This double signatures mechanism improves the security of the private ledger. When the custody event is received, it is checked. If the event is accepted, the node will save the custody event into its local database. The node also calculates and stores the public version of the event consisting of the hash value of the private event in the local copy of the public ledger. The public version and the private version of the custody event share the same *EventID*. The event generator is also responsible for sharing the public version of the event with non-trading partners.

A monitoring event is generated by an external monitor. External monitors do not have any information about the shipment except *truckID* and geolocation. The information in the monitoring event is validated through crowd-sourcing since multiple monitors are expected to report the geolocation of a single truck.

The second validation is with respect to blocks. In the proposed system, a node can propose a candidate block. In order to limit the number of proposed blocks that a node can put forward, the node needs to find a nonce that satisfies a hash value with a given format. For example, the hash value may include a number of leading zeros making the process of computing a hash value for a block computationally expensive. This strategy prevents nodes from submitting invalid blocks because other nodes in the network can refuse the proposed block. Furthermore, it prevents changes to the blockchain, since recalculating nonces for the entire chain is computationally prohibitive. After the proposed block is received, other nodes validate the events in the block. They accept the block only if the *nonce* is satisfied and all the public events have *EventHash* identical to the ones stored in the local database of the node.

3.7 Processes

The proposed system has three main processes that manage events and blocks. These processes are:

- *Private Event Process* which supports the sharing of private events among trading partners in a shipment.
- *Public Event Process* which allows the exchange and the posting of public events to the public ledger.
- *Building Block Process* which combines multiple events into a block and validates each block in the public ledger.

3.7.1 Private Event Process

The genesis event and the custody event information is encapsulated into the *PrivateGCEvents* structured message which is managed by the private event process. An administrative node generates a genesis event by extracting data from the company's ERP. A regular node triggers a custody event where the underlying message is signed by two participants. For example, when a shipment is being transferred from a supplier to a carrier, an employee of the supplier signs the custody event with his private key to confirm the event. Subsequently, an employee of the carrier signs the custody event and sends the event to the trading partners. Private events are only shared among trading partners. Therefore, a monitor will never receive a private event. As shown in Fig. 3.20, node A and node B represent an administrative node and a regular node, respectively. Once the message is received from the ERP, node A generates a unique ID for the event. If the event does not originate from the ERP, it is already associated with a unique ID. This unique ID is generated by the MongoDB database or its driver. The ID has 12 bytes where the first four bytes refer to the seconds, the next three bytes are the machine identifier followed by two bytes that represent the process id and the last three bytes are a random counter value. The administrative node will then broadcast the event to the other partners. This process is also applicable to custody events. The next step in the process consists of generating the public event associated with the private event. The hash value of the private event is computed by the event generator and the resulting public event is sent to non-trading nodes. The process concludes by having the event generator store the public event in its local copy of the public ledger.

3.7.2 Public Event Process

A public event can be one of two types (Fig. 3.21). The first type is the hash value of a private genesis or custody event. The second type is a monitoring event. Any participant in the network is able to send or receive a public event. The public events

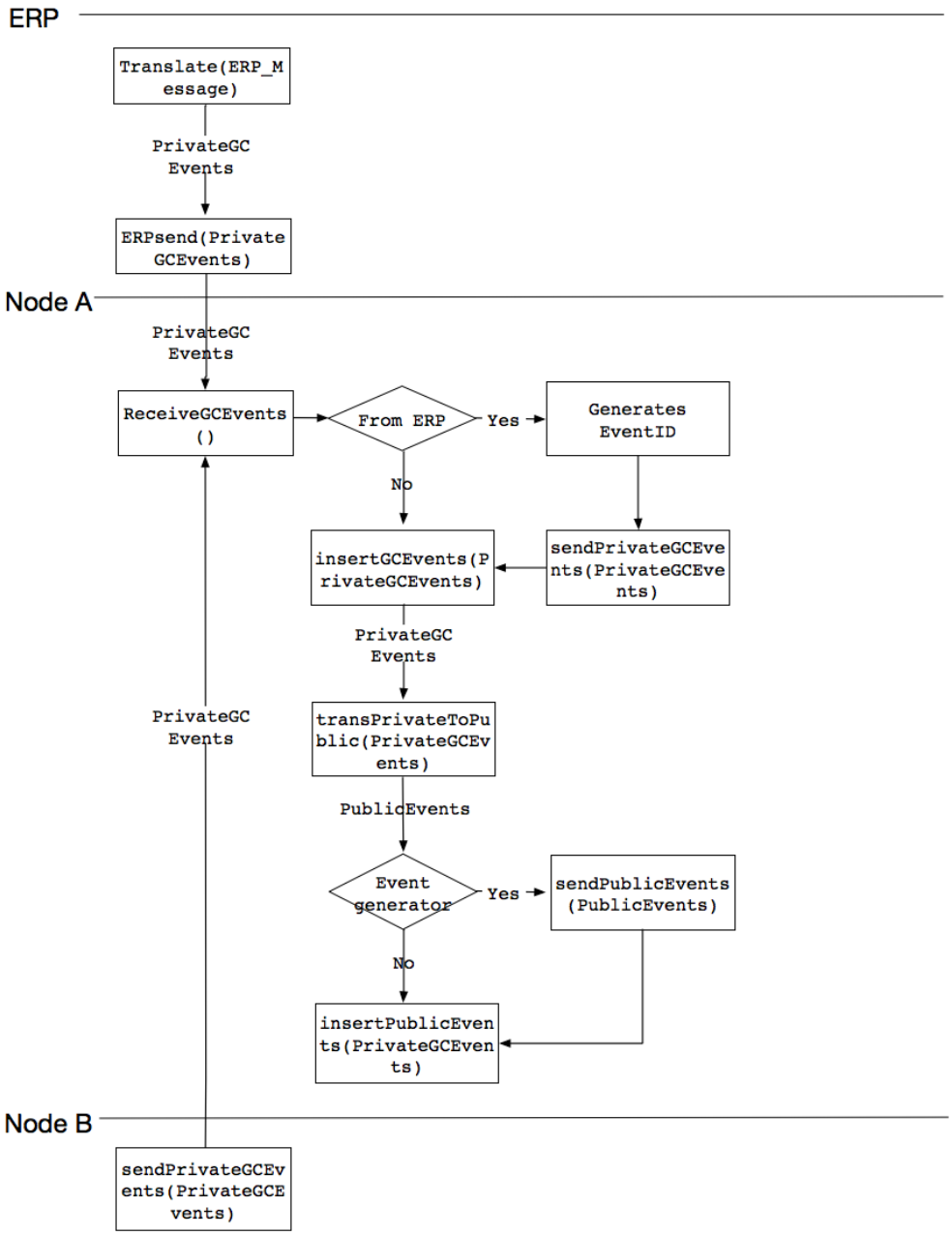

Fig. 3.20. Private Event Process

received are chained together in order to build the public ledger which is maintained by all participants.

In the case of a monitoring event, data is created when the monitor and the truck exchange a message indicating physical proximity. The monitor generates a unique ID for the event. It will also calculate the hash of the event and store it in the *EventHash* field of the *MonitorData* data structure. Finally, the monitor broadcasts the event to the participants in the public ledger.

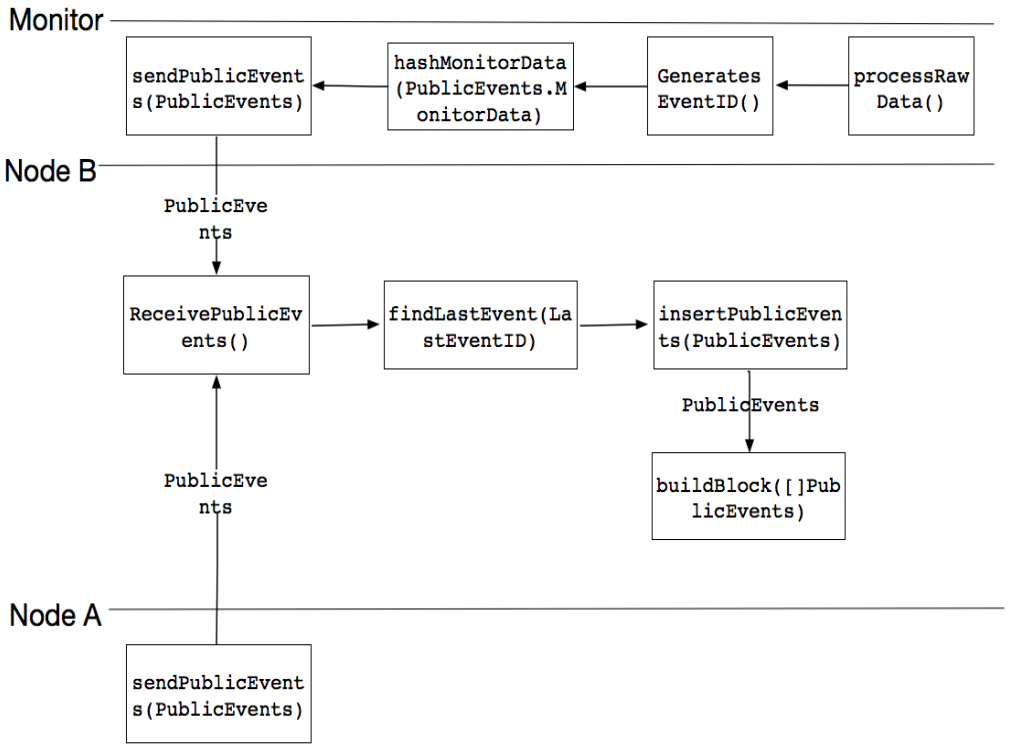

Fig. 3.21. Public Event Process

In case of the public event derived from a private genesis or custody event, the event generator calculates the hash value of the event and sends it to the non-trading partners. This approach prevents the same event from being sent multiple times.

Once a public event is received, the last inserted event in the *tempPublicEvent* collection is retrieved. If no previous event is found, the new event becomes the first event in *tempPublicEvent*. If there is a previous event in the collection, the *CurEveHash* of the event is used as the *PreEveHash* in the new event. The *CurEveHash* of the new event is calculated based on the information in the *PublicEvents* data

structure. The event is inserted into the local copy of the public ledger. Every time a public event inserted, the number of events in the *tempPublicEvent* collection is checked. If the number exceeds a preset value, a new block is built by combining multiple events. This process is discussed next.

There are three hash values in the public event data structure: *EventHash*, *PreEveHash* and *CurEveHash*. *EventHash* is calculated by the event generator. *PreEveHash* is retrieved from the previous event in the node's local database. *CurEveHash* is calculated by each node in the network including monitors based on the information in the *PublicEvents* data structure. Since each node may receive public events in different order, the *PreEveHash* of the events may differ which results in different values of the *CurEveHash*. However, the *EventHash* is independent of the order in which the events are received. Furthermore, it is consistent since it is computed by the event generator. The *Building Block Process* which is described next is responsible for receiving the block and checking the validity of each *EventHash*. If the block is accepted by the recipient node, the underlying order of the events is also accepted and the local ordering of the events in the node is discarded.

3.7.3 Building Block Process

Any node or monitor can propose a block (Fig. 3.22). A block is built by using the events in the order they are received by the node. If a nonce is found, the node broadcasts its proposed block to other nodes. When the other nodes receive the candidate block, they can check the validity of the block by recomputing the *CurBloHash*. It the *CurBloHash* meets the value received with the nonce provided, the nodes check each events' *EventHash* value in the block by comparing it to the values in the local database. If the result is different, the proposed block will be discarded. If the candidate block is confirmed, the *CurBloHash* field of the previous block is compared to the current block's *CurBloHash*. The block is stored and chained in the local database, if these two fields match. Otherwise the candidate block will be

save into a temporary collection called *tempBlocks*. The node may be missing a block or there could be multiple branches of the public ledger. Resolving these collision issues is the subject of future work.

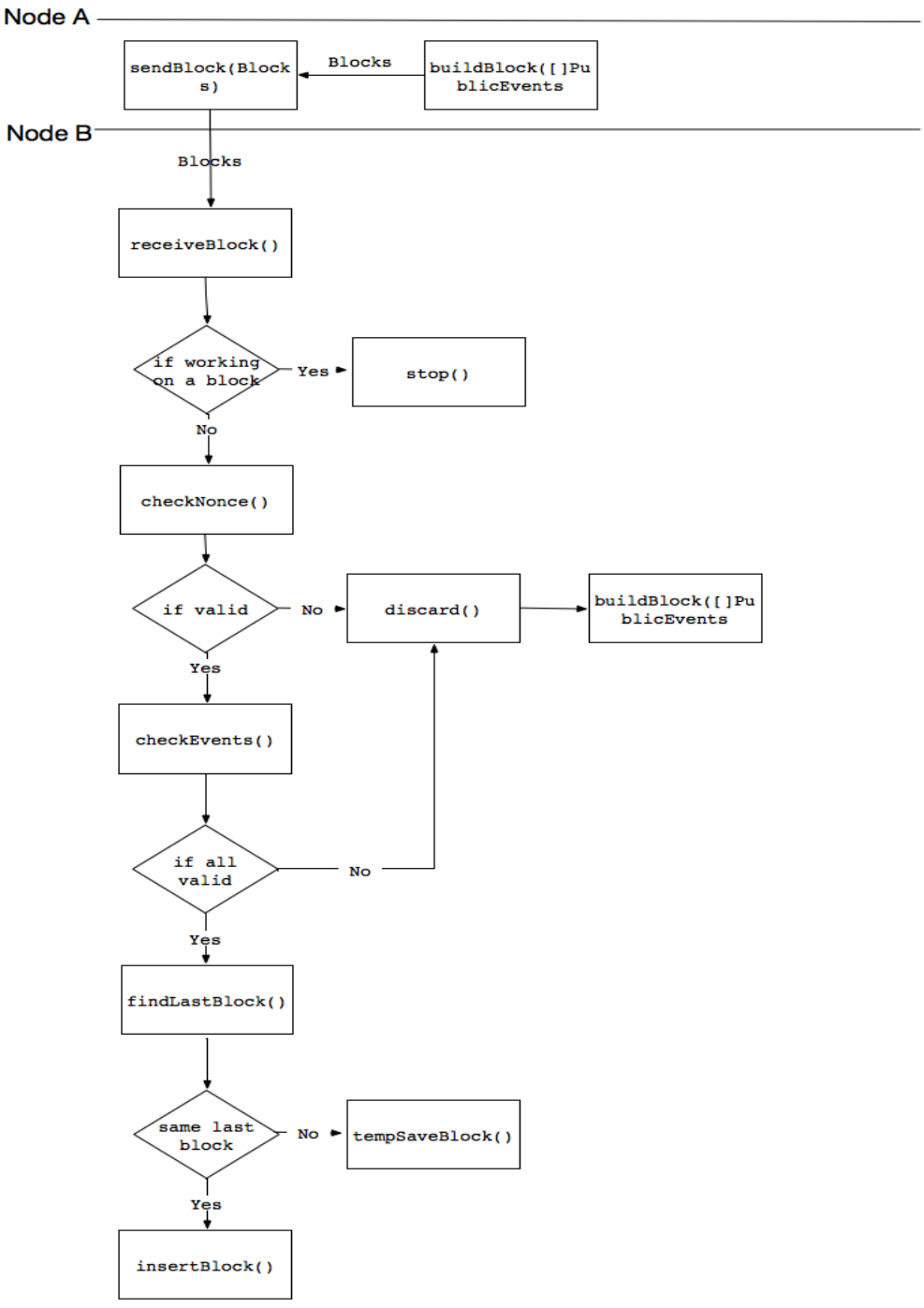

Fig. 3.22. Building Block Process

4. IMPLEMENTATION

In this chapter, the implementation of the components of the proposed system is presented specifically, the middle tier and data tier of the index server, nodes and external monitors are introduced and the chaining of the events and blocks in the public ledger blocks are described.

4.1 Index Server

The index server participates in two different functions: *QueryByCode()* and *UpdateTime()*. The first is listening for incoming requests for an IP address query for a target node from a source node. The *QueryByCode()* function shown in Algorithm 4.1 is used to find the IP address of the target node. This function takes the node's ID as the query condition and returns a document if a match is found. The returned document has the structure defined by the *NodeInfo* structure shown in Fig. 3.1

```
   Function QueryByCode(ID string)
1  |  session, err := mgo.Dial(databaseAddr)
2  |  errhandler(err)
3  |  var result NodeInfo
4  |  c := session.DB(databaseName).C(collectionName)
5  |  c.FindId(bson.ObjectIdHex(hexID)).One(result)
6  |  return result
```

Algorithm 4.1. Query Node's IP address

```
   Function UpdateTime(ID string, time string)
1  |  var new NodeInfo
2  |  var old NodeInfo
3  |  old = queryByCode(hexID)
4  |  new = old
5  |  new.timestamp = time
6  |  session, err := mgo.Dial(databaseAddr)
7  |  errhandler(err)
8  |  d := session.DB(database).C(collection)
9  |  d.Update(old,new)
10 |  return
```

Algorithm 4.2. Update Node Information

The *UpdateTime()* (Algorithm 4.2) function is invoked with the ID of the node issuing a heartbeat message. This function first calls the *QueryByCode()* function to find the old document, then it updates the time and IP address associated with the given ID in the index server's local database. This process maintains the node's IP address up-to-date.

4.2 Nodes and External Monitors

There are three kinds of nodes in the proposed system: X86 nodes running on X86 platforms, mobile nodes running on Android platforms and administrative nodes which are also running on X86 platforms but have extended functionalities. These nodes may participate in the private ledger, the public ledger or both.

The X86 nodes and administrative nodes have the the same architecture consisting of three tiers: presentation tier, middle tier and data tier. The monitors only have a middle tier and a data tier. The middle tier of an X86 node application is implemented using Golang and Mgo (MongoDB driver for Golang). The monitors can receive signals from sensors, generate and broadcast public events, update their local database and build blocks. The X86 nodes and administrative nodes also have these functionalities in addition to handling incoming user requests from the presentation tier and generating and sending private events to their partners.

The mobile node also has a three-tier architecture. The presentation tier and the middle tier are developed by using JAVA, Android studio and XML. Couchbase Lite [20], a noSQL database similar to MongoDB, is used for the data tier. The Couchbase Lite database consists of set of buckets where each bucket stores a list of documents. The JAVA application accesses the database through the Couchbase lite driver.

Most of the functionalities of the mobile node are the same as the X86 node. The user interacts with the application via the presentation tier. The middle tier is responsible for sending/receiving shipment information updates to/from other nodes as well as for updating the local database. However, the mobile device application needs additional functionalities since it is also used to acquire information from the field. The mobile nodes do not propose a block, since finding a nonce is a compute intensive process. They only receive candidate blocks and verify them.

For example, when the supplier transfers the shipment to the loading area for pick-up by the carrier, the mobile node application scans the shipment information by using, for instance, a QR code scanner. Once the information is read, the mobile node will send the updated information to all the nodes that are involved in the shipment.

The presentation tier in the administrative node is an HTTP server, the middle tier is implemented by using Golang and the data tier is implemented by using MongoDB. All the events messages that the administrative node receives are in JSON format and can be stored directly into the MongoDB database without any additional parsing. The administrative node provides remote database query functionalities for the mobile nodes that have missed some events because they dropped out of the network.

4.3 Processes

The three processes discussed in the previous chapter are responsible for building the two ledgers: the private ledger and the public ledger. The genesis events and

custody events are stored in the private ledger in the form of documents without chaining. The hash values of genesis events and custody events, along with the monitoring events are chained together in a unified data structure called *PublicEvents*. This data structure constitutes the public ledger. Since every node in the network keeps a copy of the public ledger, the events in the private ledger can be validated by checking their hash value against the event's *EventHash* field in the public ledger.

4.3.1 Private Event Process

The private event process starts with a TCP connection established through a *Listen()* (Algorithm 4.3). Once a genesis or a custody event is sent from a trading partner node, the target node accepts the connection by using the *Accept()* function (Algorithm 4.4).

```
   Function Listen(port string)
1  |  ln, err := net.Listen("tcp", port)
2  |  errhandler(err)
3  |  for do
   |  |  Accept1(ln, port)
   |  end
4  |  return
```

Algorithm 4.3. Listen Function

```
   Function Accept(ln net.Listener, port string)
1  |  lnc, err := ln.Accept()
2  |  errhandler(err)
3  |  go ReceiveGCEvents(ln, port)
4  |  return
```

Algorithm 4.4. Accept Connection

The private event is received by calling the *ReceiveGCEvents()* function in Algorithm 4.5. This function saves the event in node's local database by using the

insertGCEvents() function (Algorithm 4.6). An *EventID* is uniquely assigned to each event by the event generator. This ID is used in the two ledgers. If the node is the event generator, it also calls Algorithm 4.7 to send the private event to the trading partners.

```
   Function ReceiveGCEvents(nc net.Conn, port string)
1  |  var msg [ ]byte
2  |  var mPrivateGCEvents PrivateGCEvents
3  |  err := json.NewDecoder(nc).Decode(&msg)
4  |  errhandler(err)
5  |  e := json.Unmarshal(msg,&mPrivateGCEvents)
6  |  errhandler(e)
7  |  defer nc.Close()
8  |  if port == ":9999" then
9  |  |  mPrivateGCEvents.EventID=bson.NewObjectId()
   |  end
10 |  insertGCEvents(mPrivateGCEvents,port)
11 |  return
```

Algorithm 4.5. Receive Private Events

The next step executes Algorithm 4.8, *TransPrivateToPublic*, which translates the private event into a public event for posting to the public ledger. This function takes the same *EventID* associated wtih the private event. It also calls *hashPrivateEvent* (Algorithm 4.9) to compute the *EventHash* field of the event.

```
   Function insertGCEvents(in PrivateGCEvents, port string)
1  |  session, err := mgo.Dial("localhost")
2  |  errhandler(err)
3  |  defer session.Close()
4  |  session.SetMode(mgo.Monotonic, true)
5  |  d := session.DB("node").C("PrivateEvents")
6  |  err = d.Insert(&in)
7  |  errhandler(err)
8  |  if port == ":9999" then
9  |  |  sendPrivateGCEvents(IP, in, "9997")
   |  end
10 |  transPrivateToPublic(result,port)
11 |  return
```

Algorithm 4.6. Insert Private Events

```
   Function sendPrivateGCEvents(ipaddr string, mPrivateGCEvents
    PrivateGCEvents, port string)
1  |  c, err := net.Dial("tcp", ipaddr+":"+Port)
2  |  errhandler(err)
3  |  b,e := json.Marshal(mPrivateGCEvents)
4  |  errhandler(e)
5  |  e2 := json.NewEncoder(c).Encode(b)
6  |  errhandler(e2)
7  |  c.Close()
8  |  return
```

Algorithm 4.7. Send Private Events

The translated event is sent to other nodes by the event generator node. The function *HashPrivateEvent* takes *PrivateGCEvents* as an input, transfers the JSON message to byte code and uses SHA-256 to hash the message. It returns a byte array.

```
Function transPrivateToPublic(mPrivateGCEvents PrivateGCEvents, port string)
1   var mPublicEvents PublicEvents
2   mPublicEvents.EventID=mPrivateGCEvents.EventID
3   mPublicEvents.Timestamp=mPrivateGCEvents.Timestamp
4   mPublicEvents.EventHash=hashPrivateEvent(mPrivateGCEvents)
5   if port == ":9999" then
6   |   sendPublicEvents(IP, in, "9996")
    end
7   insertPublicEvents(mPublicEvents)
8   return
```

Algorithm 4.8. Translate Private Events to Public Events

```
Function hashPrivateEvent(mPrivateGCEvents PrivateGCEvents)[32]byte
1   bytes, err := json.Marshal(mPrivateGCEvents)
2   errhandler(err)
3   sum := sha256.Sum256(bytes)
4   return sum
```

Algorithm 4.9. Hash Private Event

4.3.2 Public Event Process

In order to receive a public event, both *Listen()* and *Accept()* are called. *ReceivePublicEvents* (Algorithm 4.10) is called from within the *Accept()* function. If the node is a monitor, the *ReceivePublicEvents* function assigns a unique ID to the monitoring event and computes an *EventHash* based on the *MonitorData* by using Algorithm 4.11. The monitor will also broadcast the events to all nodes in the network. This step is similar to the one used to broadcast a translated private event. If the current node is the event generator, it sends the public event by calling Algorithm 4.12.

```
   Function ReceivePublicEvents(nc net.Conn, port string)
1  |  var msg [ ]byte
2  |  var mPublicEvents PublicEvents
3  |  err := json.NewDecoder(nc).Decode(&msg)
4  |  errhandler(err)
5  |  e := json.Unmarshal(msg,&mPublicEvents)
6  |  errhandler(e)
7  |  defer nc.Close()
8  |  errhandler(err)
9  |  if port == ":9998" then
10 |  |  mPublicEvents.EventID=bson.NewObjectId()
11 |  |  mPublicEvents.EventHash=
12 |  |    hashMonitorData(mPublicEvents.MonitorData)
13 |  |  sendPublicEvents(IP, mPublicEvents, "9996")
   |  end
14 |  insertPublicEvents(mPublicEvents)
15 |  return
```

Algorithm 4.10. Receive Public Events

```
   Function hashMonitorData(mMonitorData MonitorData)[32]byte
1  |  bytes, err := json.Marshal(mMonitorData)
2  |  errhandler(err)
3  |  sum := sha256.Sum256(bytes)
4  |  return sum
```

Algorithm 4.11. Hash Monitor Data

```
Function sendPublicEvents(ipaddr string, mPublicEvents PublicEvents, port string)
1   c, err := net.Dial("tcp", ipaddr+":"+Port)
2   errhandler(err)
3   b,e := json.Marshal(mPublicEvents)
4   errhandler(e)
5   e2 := json.NewEncoder(c).Encode(b)
6   errhandler(e2)
7   c.Close()
8   return
```

Algorithm 4.12. Send Public Events

The final step consists of inserting the public event into the nodes's local database by calling *insertPublicEvents()*. The function (Algorithm 4.13) checks if the *tempPublicEvent* collection is empty. If there are no events in the *tempPublicEvent* collection, the received public event will be the first event in a block. The *PreEveHash* field has the same value as *EventHash*'s. If there is more than one event in the collection, the function finds the last inserted event by using the *LastEventID*. This is a global variable that keeps track of the last event inserted. The *CurEveHash* of the last event is used as *PreEveHash* of the current event. This action chains the two events together. *InsertPublicEvents()* also checks the number of events in the *tempPublicEvent* collection. Once the number exceeds a preset value, the function *buildBlock()* is called. The events that will form the new block will be extracted from the *tempPublicEvent* collection. The process of building a block is discussed next.

```
   Function insertPublicEvents(mPublicEvents PublicEvents)
1  |  var results []PublicEvents
2  |  var result PublicEvents
3  |  session, err := mgo.Dial("localhost")
4  |  errhandler(err)
5  |  defer session.Close()
6  |  session.SetMode(mgo.Monotonic, true)
7  |  c := session.DB("node").C("tempPublicEvent")
8  |  err = c.Find(nil).All(&results)
9  |  errhandler(err)
10 |  if len(results) == 0 then
11 |  |  mPublicEvents.PreEveHash=mPublicEvents.EventHash
12 |  |  LastEventID=mPublicEvents.EventID
   |  else
13 |  |  c.FindId(bson.ObjectId(LastEventID)).One(&result)
14 |  |  LastEventID=mPublicEvents.EventID
15 |  |  mPublicEvents.PreEveHash=result.CurEveHash
   |  end
16 |  PreEveHash=mPublicEvents.PreEveHash
17 |  mPublicEvents.CurEveHash=hashPublicEvent(mPublicEvents)
18 |  err = c.Insert(&mPublicEvents)
19 |  errhandler(err)
20 |  num,_:=c.Find(nil).Count()
21 |  i:=0
22 |  if num¿presetValue then
23 |  |  buildBlock(results)
24 |  |  for range results do
25 |  |  |  c.Remove(bson.ObjectIdHex(results[i].EventID))
26 |  |  |  i++
   |  |  end
   |  end
27 |  return
```

Algorithm 4.13. Insert Public Events

```
   Function hashPublicEvent(mPublicEvents PublicEvents)[32]byte
1  |  bytes, err := json.Marshal(mPublicEvents)
2  |  errhandler(err)
3  |  sum := sha256.Sum256(bytes)
4  |  return sum
```

Algorithm 4.14. Hash Public Event

4.3.3 Building Block Process

The *buildBlock* (Algorithm 4.15) function has similar functionalities to that of the function *insertPublicEvents*. It checks the *Blocks* collection to see if it is empty and initializes the first block if needed. The block is built locally by hashing (Algorithm 4.16) and then sending (Algorithm 4.17) the block to other nodes. However, the block is considered as a candidate block as long as it is not approved by other nodes. The validation of a block across all nodes in the network including resolving collisions and multiple branches in the public ledger is the subject of future work.

```
   Function buildBlock(results []PublicEvents)
 1 |  var mBlock Blocks
 2 |  var resultss []Blocks
 3 |  var result Blocks
 4 |  mBlock.BlockID=bson.NewObjectId()
 5 |  mBlock.Timestamp=time.Now().UTC().String()
 6 |  a:=make([]PublicEvents, len(results))
 7 |  mBlock.Event=a
 8 |  i:=0
 9 |  for range results do
10 |  |  mBlock.Event[i]=results[i]
11 |  |  i++
   |  end
12 |  session, err := mgo.Dial("localhost")
13 |  errhandler(err)
14 |  defer session.Close()
15 |  session.SetMode(mgo.Monotonic, true)
16 |  c := session.DB("node").C("Blocks")
17 |  err = c.Find(nil).All(&resultss)
18 |  errhandler(err)
19 |  if len(resultss) == 0 then
20 |  |  mBlock.PreBloHash=hashBlcok(mBlock)
21 |  |  LastBlockID=mBlock.BlockID
   |  else
22 |  |  c.FindId(bson.ObjectId(LastBlockID)).One(&result)
23 |  |  LastBlockID=mBlock.BlockID
24 |  |  mBlock.PreBloHash=result.CurBloHash
   |  end
25 |  var bHash [32]byte
26 |  setHash := make([]byte, 32)
27 |  found:=false
28 |  mBlock.Nonce=0
29 |  for found!=true do
30 |  |  bHash=hashBlcok(mBlock)
31 |  |  if bytes.Compare(bHash[:], setHash[:])==-1 then
32 |  |  |  found=true
   |  |  end
33 |  |  mBlock.Nonce++
   |  end
34 |  mBlock.CurBloHash=bHash
35 |  err = c.Insert(&mBlock)
36 |  errhandler(err)
37 |  sendBlock(mBlock)
38 |  return
```

Algorithm 4.15. Build Block

```
Function hashBlcok(mBlocks Blocks)[32]byte
1 | bytes, err := json.Marshal(mBlocks)
2 | errhandler(err)
3 | sum := sha256.Sum256(bytes)
4 | return sum
```

Algorithm 4.16. Hash Blocks

```
Function sendBlock(ipaddr string, mBlocks Blocks, port string)
1 | c, err := net.Dial("tcp", ipaddr+":"+Port)
2 | errhandler(err)
3 | b,e := json.Marshal(mBlocks)
4 | errhandler(e)
5 | e2 := json.NewEncoder(c).Encode(b)
6 | errhandler(e2)
7 | c.Close()
8 | return
```

Algorithm 4.17. Send Candidate Block

4.4 Testing

The test scenario discussed in this chapter demonstrates how both the public ledger and the private ledgers are constructed. The testing environment consists of three nodes and one monitor. The testing focuses on the exchange between the nodes. For clarity, exchanges between the nodes and the index server are skipped and the emphasis is placed on the public and the private ledgers. Node A in the test scenario is an administrative node which belongs to company A. Node B is a node which belongs to company B and node C is a node which belongs to company C. The following five events are considered in the test scenario:

- Event 1: A genesis event that is generated by node A. It is an event related to order 1 in which company A and company B are participating.

- Event 2: A custody event that is sent by node B. This is an event related to order 2 which is being shared with company B and company C.
- Event 3: A monitoring event that is generated by an external monitor. It consists of a public event that is sent to all nodes in the network. It represents a geolocation update for order 2.
- Event 4: A custody event that is generated by node B. This is an event related to order 1 in which company A and company B are participating.
- Event 5: A monitoring event that is generated by an external monitor. It consists of a public event that is sent to all nodes in the network and represents a geolocation update for order 1.

Two sub-networks are created by the above test scenario. One is shared by node A and node B for order 1. The other sub-network is shared by node B and node C for order 2. These subnetworks are private to the trading partners. A global network is also needed to support the exchange of public events that form the public ledger.

A simulator was created to trigger the five events mentioned above. For a genesis event, the simulator mimics an ERP which is responsible for sending the genesis event information to an administrative node. For a custody event, the simulator verifies and signs the event. For a monitoring event, the simulator acts as a sensor that sends geolocation data to the external monitors. The simulator uses TCP port "9999" to communicate with nodes and port "9998" to communicate with monitors.

```
Select a number to trigger an event:
1: Node A generates a genesis event which is shared with node B
2: Node B generates a custody event which is shared with node C
3: Monitor A generates a monitoring event
4: Node B generates a custody event which is shared with node A
5: Monitor A generates a monitoring event
Enter selection is:
```

Fig. 4.1. Steps of the Test Scenario

The simulator is used to send event 1 (Fig. 4.1) which is received by node A. Then node A assigns a unique ID for this genesis event. Since this is a private event, it is only sent to node B. Node A will then insert this event into its local database in the *PrivateEvents* collection. Node A is considered as the event generator. Therefore, it will also translate the genesis event to a public event and compute the *EventHash*.

The translated public event is broadcasted to all the nodes in the entire network by node A. Node B receives the detailed genesis event from node A and save it to its local database. Node C and the monitor receive the public event and insert it into the *tempPublicEvent* collection after chaining and calculating *CurEveHash*. Since this is the first event in the database, the *PreEveHash* field has the same hash value as *EventHash*. There is a counter in *insertPublicEvents()* function. If this counter reaches a preset value, *buildBlock()* function is called to construct a block. To demonstrate this functionality in the test scenario, this value is set to 4. Therefore, when there are more than four public events in the *tempPublicEvent* collection, these events are used to form a candidate block. Fig. 4.2 and Fig. 4.3 show the *PrivateEvents* collection and the *tempPublicEvent* collection in Node A after it receives event 1, respectively. There is one document in the *PrivateEvents* collection. This private event is shared by node A and B. Therefore, node B also has the same document in its *PrivateEvents* collection. For all nodes and the monitors, the translated event is stored in the *tempPublicEvent*. Given this is the first event in the public ledger, all *tempPublicEvent* collections across all nodes (Fig. 4.3) are same.

This simulator is used next to trigger event 2. Subsequently, node B receives the custody event. Node B assigns a unique ID to the event and executes the same steps executed by node A. Node A does not participate in event 2. Therefore, it does not receive the private event and the private ledger of node A remains unchanged (Fig. 4.2). However, node A receives the corresponding public event as any other node in the network. This is shown in Fig. 4.4.

```
> db.PrivateEvents.find()
{ "_id" : ObjectId(‘‘595514e51d41c826a2b0894b"),
  "timestamp" : "2017-1-1 10:00PM",
  "eventdetails" :
     { "status" : "Order Placed",
       "gpscordsx" : "", "gpscordsy" : ""
     }
}
```

Fig. 4.2. PrivateEvents Collection of Node A after Event 1 is Received

```
> db.tempPublicEvent.find()
{ "_id" : ObjectId(‘‘595514e51d41c826a2b0894b"),
  "timestamp" : "2017-1-1 10:00PM",
  "monitordata" :
  { "monitorid" : "",
    "truckid" : "",
    "geoinfo" :
      { "gpscordsx" : "",
        "gpscordsy" : ""
      }
   },
   "eventhash" : BinData(0,"/IRbc+0y6HvQPHAKwQPIxxGlFAXkVe2WB1OXW
                 QzSe/I="),
   "preevehash" : BinData(0,"/IRbc+0y6HvQPHAKwQPIxxGlFAXkVe2WB1OX
                  WQzSe/I="),
   "curevehash" : BinData(0,"A6NwbhE4Gix7quTB+g1uCrNu/agoTDJbJkk0
                  QZf7EV0=")
}
```

Fig. 4.3. tempPublicEvent Collection of Node A after Event 1 is Received

In step 3, the simulator is used to trigger event 3 which is a monitoring event. Monitoring events are public events. The monitor assigns a unique ID to the public event and computes the *EventHash*. It saves the events into the *tempPublicEvent* collection and also sends it to node A, B and C. Every time a public event is stored into public ledger, the *insertPublicEvents* function is executed to retrieve the last inserted public event. The value of *CurEveHash* for this event is used as the *PreEveHash*

```
> db.tempPublicEvent.find()
{ "_id" : ObjectId(‘‘595514e51d41c826a2b0894b"),
  "timestamp" : "2017-1-1 10:00PM",
  "monitordata" :
  { "monitorid" : "",
    "truckid" : "",
    "geoinfo" :
      { "gpscordsx" : "",
        "gpscordsy" : ""
      }
   },
   "eventhash" : BinData(0,"/IRbc+0y6HvQPHAKwQPIxxGlFAXkVe2WB1OX
                 WQzSe/I="),
   "preevehash" : BinData(0,"/IRbc+0y6HvQPHAKwQPIxxGlFAXkVe2WB1O
                  XWQzSe/I="),
   "curevehash" : BinData(0,"A6NwbhE4Gix7quTB+g1uCrNu/agoTDJbJkk
                  0QZf7EV0=")
}
{ "_id" : ObjectId(‘‘595515221d41c811406c398f"),
  "timestamp" : "2017-3-3 04:30PM",
  "monitordata" :
  { "monitorid" : "",
    "truckid" : "",
    "geoinfo" :
      { "gpscordsx" : "",
        "gpscordsy" : ""
      }
   },
   "eventhash" : BinData(0,"5mDGKxSaXZCLmkfugYxpqbhoaIdzUJAnaQnT
                 jgsjwgo="),
   "preevehash" : BinData(0,"A6NwbhE4Gix7quTB+g1uCrNu/agoTDJbJkk
                  0QZf7EV0="),
   "curevehash" : BinData(0,"tNELVhCUwHQznOcyQ1spp3wxHtIcNqOhL/i
                  SKi4vDC0=")
}
```

Fig. 4.4. tempPublicEvent Collection of Node A after Event 2 is Received

for the current event. The *tempPublicEvent* collection of node A after the receipt of public event 3 is shown in Fig. 4.5.

```
{ "_id" : ObjectId(‘‘595515491d41c8492a5ea800"),
  "timestamp" : "2017-3-4 07:30PM",
  "monitordata" :
  { "monitorid" : "",
    "truckid" : "",
    "geoinfo" :
      { "gpscordsx" : "16",
        "gpscordsy" : "63"
      }
   },
   "eventhash" : BinData(0,"oNAFBBMrmUP36CYh5UdIqz5IAFQsNG+LwpmRR
                 B08Kcw="),
   "preevehash" : BinData(0,"tNELVhCUwHQznOcyQ1spp3wxHtIcNqOhL/iS
                  Ki4vDC0="),
   "curevehash" : BinData(0,"iQ9VPUYwZS/M26aUbZEicbs01pzoNIhkeEZQ
                  wIuATSU=")
}
```


Fig. 4.5. New Document in tempPublicEvent Collection of Node A after Event 3 is Received

Event 4 is same as event 1 except the event is sent by node B. Broadcasting event 5 is exactly same as for event 3. When event 5 is received by all nodes, the *buildBlock()* function is invoked. All nodes and monitors in the network start building blocks. A nonce is needed to build a block. The nodes including monitors in the network assign a unique ID for the new block. They find the last block in the chain, increment the nonce starting from zero until the required hash is found. This hash value is stored in *CurBloHash*. The resulting block is considered as a candidate block until it is validated by other nodes. This block is shown in Fig. 4.6 and Fig. 4.7.

```
> db.Blocks.find()
{ "_id" : ObjectId(‘‘595516071d41c826a2b0894c"),
  "event" : [
     { "_id" : ObjectId(‘‘595514e51d41c826a2b0894b"),
       "timestamp" : "2017-1-1 10:00PM",
       "monitordata" :
       { "monitorid" : "",
         "truckid" : "",
         "geoinfo" :
          { "gpscordsx" : "",
            "gpscordsy" : ""
          }
        },
       "eventhash" : BinData(0,"/IRbc+0y6HvQPHAKwQPIxxGlFAXkVe2W
                     B1OXWQzSe/I="),
       "preevehash" : BinData(0,"/IRbc+0y6HvQPHAKwQPIxxGlFAXkVe2
                      WB1OXWQzSe/I="),
       "curevehash" : BinData(0,"A6NwbhE4Gix7quTB+g1uCrNu/agoTDJ
                      bJkk0QZf7EV0=")
     },
     { "_id" : ObjectId(‘‘595515221d41c811406c398f"),
       "timestamp" : "2017-3-3 04:30PM",
       "monitordata" :
       { "monitorid" : "",
         "truckid" : "",
         "geoinfo" :
          { "gpscordsx" : "",
            "gpscordsy" : ""
          }
        },
       "eventhash" : BinData(0,"5mDGKxSaXZCLmkfugYxpqbhoaIdzUJAn
                     aQnTjgsjwgo="),
       "preevehash" : BinData(0,"A6NwbhE4Gix7quTB+g1uCrNu/agoTDJ
                      bJkk0QZf7EV0="),
       "curevehash" : BinData(0,"tNELVhCUwHQznOcyQ1spp3wxHtIcNqO
                      hL/iSKi4vDC0=")
      },
```

Fig. 4.6. Candidate Block in Node A

```
    { "_id" : ObjectId(‘‘595515491d41c8492a5ea800"),
      "timestamp" : "2017-3-4 07:30PM",
      "monitordata" :
      { "monitorid" : "",
        "truckid" : "",
        "geoinfo" :
         { "gpscordsx" : "16",
           "gpscordsy" : "63"
         }
       },
      "eventhash" : BinData(0,"oNAFBBMrmUP36CYh5UdIqz5IAFQsNG+L
                    wpmRRBO8Kcw="),
      "preevehash" : BinData(0,"tNELVhCUwHQznOcyQ1spp3wxHtIcNqO
                     hL/iSKi4vDC0="),
      "curevehash" : BinData(0,"iQ9VPUYwZS/M26aUbZEicbs01pzoNIh
                     keEZQwIuATSU=")
     },
     { "_id" : ObjectId(‘‘5955156c1d41c811406c3990"),
       "timestamp" : "2017-4-4 04:30PM",
       "monitordata" :
        { "monitorid" : "",
          "truckid" : "",
          "geoinfo" :
            { "gpscordsx" : "",
              "gpscordsy" : ""
             }
       },
      "eventhash" : BinData(0,"WYNtQT18tZJmIk14D7n9EzH8Ob2q/KFa
                    ylKNyXpGcSM="),
      "preevehash" : BinData(0,"iQ9VPUYwZS/M26aUbZEicbs01pzoNIh
                     keEZQwIuATSU="),
      "curevehash" : BinData(0,"s1sji/sBHv5JNN8IKoH9ZaWh4dFZH16
                     9sQKS44JVbXs=")
     } ],
  "timestamp" : "2017-06-29 15:00:23.512742763 +0000 UTC",
  "nonce" : 8,
  "preblohash" : BinData(0,"l+hAiPl/vyaY31rqxYijAZ0GxJLd0Kyy+lgZ
                   OF/adrE="),
  "curblohash" : BinData(0,"HSo7KblHONS73Y4clZlqBFE1AhxAxbDOexPA
                   rcUG3SY=")
}
```


Fig. 4.7. Candidate Block in Node A cont.

5. CONCLUSION

This thesis presents a framework that supports the timely delivery of field information during the physical distribution phase of supply chain. The proposed solution is scalable and takes into consideration the privacy and validity of information being exchanged. Moreover, because it is cost effective, the solution enables small, medium as well as large businesses to interact in a dynamic and shipment-centric manner through a private ledger that digitizes the transfer of each shipment from source to destination. Information in the private ledger can be verified through a public ledger which acts as a system of record for all events as well as maintain monitoring events. The monitoring events reflect the movement of the shipment in real time. Both third party monitors and nodes are engaged in the validation of the events in the public ledger.

A scalable data model and several advanced data structures were designed to support the information exchange among the various participants in the network including the index server, nodes and external monitors. These data structures are based on the EDI-214 standards and therefore support the interoperability of the proposed system with existing supply chain information systems.

This thesis focuses on the middle tier and data tier of the proposed framework. The blockchain technology is used to enable supply chain visibility by combining a set of distributed databases. These databases implement a public ledger which contains an immutable business record for all the events in the physical distribution phase of supply chain. Custody events have corresponding entries in the public ledger. These entries consist of the *EventID* and the *EventHash* value. The validity of the custody events in the private ledger can be verified by comparing them to the correspond-

ing entries in the public ledger. The monitoring events are chained with the public version of the genesis events and custody events in the pubic ledger. This approach improves the validity of the information in the public ledger.

There are several areas for future extension to the proposed framework. These include:

- Block verification: Several enhancement to the block verification can be considered. First, the difficulty of the computation of the hash value for a block needs to be investigated and the number of events in a block needs to be adjusted accordingly. In addition, mechanisms should be developed to handle block collision resulting from multiple blocks being received at the same time. In Bitcoin, duplicate blocks are also chained to the blockchain and each branch is allowed to grow as more blocks are received. The nodes in the network only consider the longest blockchain. Overtime, only the longest chain will survive.

- Monitoring event verification: Shipment location information is collected by third-party monitors. Relying on a single monitoring event may not infer sufficient reliability. Multiple monitoring events are needed. In fact, the level of trust of the public ledger is directly proportional to the number of monitors involved in the network and the number of sighting of a given truck. Analyzing the trust level of the public ledger based on the number of monitoring events is an area for future work.

In conclusion, the proposed framework is a cost-efficient and scalable solution for supply chain visibility. Even though the system is still in the prototype phase, it demonstrates the ability of the framework to deliver field information to all stakeholders in pseudo real-time. The blockchain technology is a current trend in supply chain system. We believe that the proposed architecture is a practical solution that takes into consideration business requirements.

REFERENCES

REFERENCES

[1] K. L. Croxton, S. J. Garcia-Dastugue, D. M. Lambert, and D. S. Rogers, "The supply chain management processes," *The International Journal of Logistics Management*, vol. 12, no. 2, pp. 13–36, 2001.

[2] A. Leon, *Enterprise resource planning*. McGraw-Hill Education, 2014.

[3] R. Sudarsan, S. J. Fenves, R. D. Sriram, and F. Wang, "A product information modeling framework for product lifecycle management," *Computer-aided design*, vol. 37, no. 13, pp. 1399–1411, 2005.

[4] M. Barratt and A. Oke, "Antecedents of supply chain visibility in retail supply chains: a resource-based theory perspective," *Journal of operations management*, vol. 25, no. 6, pp. 1217–1233, 2007.

[5] C. L. Iacovou, I. Benbasat, and A. S. Dexter, "Electronic data interchange and small organizations: Adoption and impact of technology," *MIS quarterly*, pp. 465–485, 1995.

[6] D. Crockford, "The application/json media type for javascript object notation (json)," 2006 (accessed June 10, 2017). [Online]. Available: https://tools.ietf.org/html/rfc4627.

[7] K. Chodorow, *MongoDB: the definitive guide*. " O'Reilly Media, Inc.", 2013.

[8] S. Li and B. Lin, "Accessing information sharing and information quality in supply chain management," *Decision support systems*, vol. 42, no. 3, pp. 1641–1656, 2006.

[9] R. Karpinski, "E2open at one," *InternetWeek, August*, vol. 1, 2001.

[10] F. Andera and D. Derringer, "("systems, applications, products in data processing") sap: Implications for computer information systems," *Journal of Computer Information Systems*, vol. 39, no. 1, pp. 72–75, 1998.

[11] J. Holmström, "Business process innovation in the supply chain–a case study of implementing vendor managed inventory," *European journal of purchasing & Supply Management*, vol. 4, no. 2-3, pp. 127–131, 1998.

[12] M. Parfett, *What is EDI?: a guide to electronic data interchange*. Blackwell Pub, 1992.

[13] M. Swan, *Blockchain: Blueprint for a new economy*. " O'Reilly Media, Inc.", 2015.

[14] L. P. Nian and D. Chuen, "Introduction to bitcoin," *Handbook of Digital Currency: Bitcoin, Innovation, Financial Instruments, and Big Data*, p. 1, 2015.

[15] N. Andrade, M. Mowbray, A. Lima, G. Wagner, and M. Ripeanu, "Influences on cooperation in bittorrent communities," in *Proceedings of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems*. ACM, 2005, pp. 111–115.

[16] G. Fox, "Peer-to-peer networks," *Computing in Science & Engineering*, vol. 3, no. 3, pp. 75–77, 2001.

[17] S. Guha and N. Daswani, "An experimental study of the skype peer-to-peer voip system," Cornell University, Tech. Rep., 2005.

[18] S. Ramamurthy, "Leveraging blockchain to improve food supply chain traceability," 2017 (accessed February 10, 2017). [Online]. Available: https://www.ibm.com/blogs/blockchain/2016/11/leveraging-blockchain-improve-food-supply-chain-traceability/.

[19] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, 2014.

[20] D. Ostrovsky and Y. Rodenski, "Couchbase lite on android," in *Pro Couchbase Server*. Springer, 2014, pp. 283–292.

[21] Z. Wei-ping, L. Ming-Xin, and C. Huan, "Using mongodb to implement textbook management system instead of mysql," in *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference*. IEEE, 2011, pp. 303–305.

[22] *The EDI interchange structure*, 2017 (accessed July 6, 2017). [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSDKJ8_6.2.0/com.ibm.wpg.entadv.doc/config/hub/edi_interchange_struct.html.

[23] M. Pilkington, "Blockchain technology: principles and applications," *Research Handbook on Digital Transformations*, 2015.