The Dissertation Committee for Douglas Howard Steves
certifies that this is the approved version of the following dissertation:

Contract in Electronic Commerce

Committee:

Mohamed G. Gouda, Supervisor

Michael D. Dahlin

Simon S. Lam

Aloysius K. Mok

J. D. Tygar

Contract in Electronic Commerce

by

Douglas Howard Steves, B.A.; B.S.; M.A.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2005

To my parents Howard and Romaline Steves,

who taught me to read and try new things;

to my dogs Soji and Laredo,

who taught me the value of perseverance and patience (respectively);

and, of course, to Vesta.

Acknowledgments

I would like to thank:

my advisor, Mohamed Gouda, for his exceptional patience;

the members of my dissertation committee for their diligence;

Chris Edmondson-Yurkanan for her considerable assistance and advice;

my friends Lauren Johnson, Bob Wescott and Kevin Brady – without their encouragement this dissertation would not have been completed.

DOUGLAS H. STEVES

The University of Texas at Austin

December 2005

Contract in Electronic Commerce

Publication No. _____

Douglas Howard Steves, Ph.D.
The University of Texas at Austin, 2005

Supervisor: Mohamed G. Gouda

Commerce is the exchange of goods, services and financial instruments conducted by procedures which define a set of properties sufficient to insure valid and effective transactions. Electronic commerce is commerce conducted using computer network protocols as the communication medium. If these network protocols have the same properties as the procedures used for non-electronic commerce, then electronic commerce will support equally valid and effective transactions.

Forum is a system for electronic commerce designed for general forms of commerce rather than specific transaction types. It includes protocols which support the contractual and structural properties of non-electronic commerce, as well as the security properties necessitated by both the value of the exchanged items and the additional risks of the communication medium.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Markets, centralized places for the exchange of goods, are a precondition for the development of commerce. They are a prominent feature of every important civilization, from the caravansarai and souks of the ancient Near East to the stock markets and financial districts of New York, London and Tokyo.

The earliest markets were probably simple places to exchange goods, probably by barter. The caravansarai (inns of commerce) were originally way stations along major trade routes that provided shelter, security and supplies for merchants transporting goods between cities. Over time, they became trading posts themselves, albeit on a small scale.

Souks were markets within cities, and for this reason represent an important step in the evolution of markets. The great cities of the Middle East – Byblos, Beirut, Damascus – were (and mostly still are) famous for their souks. An even more significant advance was the generalization of the market to include other facets of city life.

The Agora of Athens [41] was originally a meeting place for a popular political assembly. It became a public meeting place during the 6th century B.C., and by classical times also included commercial markets and religious shrines. These were all located within a small space about thirty acres in size.[1] Socrates engaged in debates in the Agora, and was tried and executed there as well.[2] Its architectural hallmark was the stoa, a covered but otherwise open space with columns forming one or more sides. The Stoics borrowed their name from the stoas where they originally gathered.

The Forum of Rome [13] was located near the Tiber river, bordered by two of the seven Roman Hills. It was as old as the Agora, but with more humble origins:

Hic ubi nunc fora sunt, udae tenuere paludes;[3]
Ovid, *Fasti* VI, 401

It was first used as a cemetery by the earliest inhabitants of Rome, who themselves lived on the surrounding hills.

[1] Or about 3/4 the size of the original Forty Acre campus of the University of Texas, bounded by Guadalupe, Speedway, and 21st and 24th Streets.

[2] Plato, having learned not to mix commerce and academics, wisely located his Academy elsewhere.

[3] Here, where the Forum is now, was a marshy swamp;

As the population increased and real estate became scarcer, the land around the Forum was "re-zoned", first for housing and then for commerce. Archaeological evidence indicates this occurred during the 6th century B.C.. Within a hundred years, the Forum had become so central to Rome that the oldest road in Rome, the *Via Sacra*, began there, and Rome's first legal code, the Twelve Tables, was published there.

The original Forum was quite small, around 200 by 300 feet, and only grew to around eight acres. Like the Agora, it contained monuments and places of public and private assembly, in addition to markets. The Roman Senate met there, as did the *comitia curiata*, the oldest of the Roman political assemblies, and, later, the *comitia tributa*. It was also there that the *praetor* posted his yearly edict, the source of much of the development of Roman contract law.

And the Forum also contained religious shrines, most notably the Temple of Vesta, who was one of the oldest of the Roman deities with origins in the ancestor worship of the early Romans. Her temple contained sacred relics and housed an eternal flame maintained by the Vestales, the priestesses of Vesta.

The importance of the Forum and the Agora was due not to their size or location, but instead to the commingling of their seemingly disparate elements. Just as a market was the generalization of a transaction, with many buyers and sellers and types of goods instead of just one of each, the Forum and the Agora represented the generalization of a market, containing all aspects of commerce: political, personal and religious, as well as economic.

In the following chapters, this dissertation describes the basic properties, main algorithm, design and implementation of Forum, a system for electronic commerce intended to enable the same synergistic interactions as its namesake.

Chapter 2

Properties

2.1 Overview

property *n.* [24]

5.a. A characteristic trait or peculiarity.

b. A special capability or power; virtue.

c. A quality serving to define or describe an object or substance.

d. A characteristic attribute possessed by all members of a class.

Commerce is the exchange of goods, services and financial instruments among individuals, businesses or governments. Exchanges can be direct or indirect, singular or manifold, instantaneous or protracted.

An instance of exchange is termed a transaction, which will serve here as the unit of commerce. A transaction can have few or many participants, each with different roles, including direct participants, such as buyers and sellers, and indirect participants like archivists, licensors, taxers and arbitrators.

Commerce is carried out by means of procedures which define the interactions among transaction participants. The basis for these procedures may be explicit in law or implicit in the types of commerce being transacted. These procedures imply a set of properties required for valid and effective transactions.

Electronic commerce is the exchange of goods, services and financial instruments using computer systems and networks. Each participant in an electronic transaction is represented by a computer process, which communicates with other processes using network protocols.

These protocols are analogous to the procedures used in non-electronic commerce, and thus they govern the interaction among the participants in an electronic transaction. If these protocols support the same properties as the procedures used in non-electronic commerce, then these electronic transactions will be equally valid and effective.

There are two types of object properties to consider: those which are inherent to an object or at least strongly implied by it (definitions 5.a, 5.c and 5.d), and those which are

desirable, either in general or for the particular object (definition 5.b).

Because every transaction is a contract, the first type must include the basic properties of legal contracts. And since the financial aspect of transactions strongly implies the need for protection, this category also includes security properties which enable access control and accountability. These will be especially important since the use of computer networks increases the opportunity for malevolent or malicious behavior.

The second type of object property is more difficult to address because of the subjective nature of virtue. Here it will be limited to structural properties which define the relationships among elements of an object, since these are applicable to most objects.

The contractual, security and structural properties have been discussed previously in [38, 39, 40]. For contract, this chapter explores the historical and analytical bases of the properties in order to provide greater justification for their inclusion; for security and structure, this chapter mostly summarizes and clarifies the earlier discussion.

The next section is about contract law: in section 2.2.1, the economic basis for contract is analyzed to understand its importance in a political economy; in section 2.2.2, the law and its evolution are discussed in order to understand the process and significance of change in a legal system; in section 2.2.3, the history and civil law of Rome are described to understand the importance of Roman contract law; in section 2.2.4, the forms of Roman contract are analyzed to determine their most salient features; in section 2.2.5, these features are used to derive the basic properties of contract; finally, in section 2.2.6 the well-known ACID properties of database transactions[14] are compared to the contractual properties of economic transactions, and are analyzed for correctness and conciseness.

Section 2.3 is about security: in section 2.3.1, a model for computer security is presented which is applicable to both storage and communication system security; in section 2.3.2, security properties for electronic commerce are defined based directly on communication system security properties. Section 2.4 uses non-electronic commerce as a structural model for electronic commerce, and section 2.5 provides a summary.

2.2 Contract

> *There are few general propositions concerning the age to which we belong which seem at first sight likely to be received with readier concurrence than the assertion that the society of our day is mainly distinguished from that of preceding generations by the largeness of the sphere which is occupied in it by Contract.* [22, p 295]

2.2.1 The Economics of Contract

Much of the law is concerned with obligations, which compel, limit or regulate the actions of persons and other legal entities. Legal obligations arise from either delicts or contracts. Delictual obligations are incurred by torts or crimes, while contractual obligations are incurred by promises.

> *Wealth in a commercial age is made up largely of promises.* [27, p 236]

Not all promises have legal significance. Which promises between individuals and organizations should be enforced by governments is mostly a philosophical question [27]. Of practical interest here is the economic rationale that the enforcement of promises related to the exchange of goods, services and money enables trade and specialization of labor, both of which are among the bases of a modern economy.

Of course, the converse is not true – the public enforcement of private promises does not imply a modern economy, but the absence of effective and efficient legal sanctions governing economic obligations will reduce the economy to ‘οι οικος - originally denoting a rudimentary economy in which production and consumption of goods and services is structured in units of one or a few households.

Trade

> *Contract is the handmaid if not actually the child of trade. Merchants and bankers must have what soldiers and farmers seldom need, the means of making and enforcing various agreements with ease and certainty.* [7, p 1]

Trade is the exchange of commodities. The first form of trade would have been barter – a direct and immediate exchange of one thing for another. A barter economy, however, limits production to those commodities that can be more or less evenly exchanged between individuals or among small groups in a constrained geographical area. The use of money as an exchange medium reduces these limits, of course, but rudimentary forms of money (e.g. specie) are subject to the same acceptability, unitary and geographical constraints as are commodities.

> *But for anything more than the most rudimentary commerce, a cash sale is not enough; what is needed is a promissory or credit sale, i.e. one in which either the payment of the price or the transfer of the thing or both are to take place at a later date. The seller or the buyer or both must be able to bind the other to the performance of a promise.* [25, p 161]

More mature financial instruments issued by banks and governments, such as currency and letters of credit, enable general forms of trade, but the acceptability of these instruments depends on the issuer's guarantees, which in turn are promises deemed enforceable by the seller who accepts the instrument in lieu of a real commodity. Thus trade is dependent on enforceable promises.

Specialization of Labor

Specialization of labor is the division of work in an economy such that individual workers are employed in the production of a few commodities, at most, and most commodities are produced by several workers. One person might farm, another might cook, and still others

might devise recipes or publish restaurant reviews. Another example is a factory assembly line on which workers successively perform different tasks to produce a single product.

Specialization allows for the increased efficiency in production primarily because each individual needs fewer skills, and thus can become more proficient at the reduced set of tasks. But greater specialization in the work force implies reduced self-sufficiency, even among extended family groups, and so it is necessary to trade either labor or the finished commodities.

Although the exchange of labor can occur directly in a barter or gift economy, these imply a close geographical or personal relationship that result in a limited, primitive economy. These limitations can be ameliorated by using money as an exchange medium in the labor market. But the use of monetary instruments, whether specie or paycheck, depends upon their acceptability, and so specialization of labor also depends upon enforceable promises.

2.2.2 The Law of Contract

The law governs the relationship among individuals and organizations, public or private. It is embodied in a set of rules which defines unlawful conduct for which the government is willing and able to provide a remedy, and which specifies the procedures by which a remedy may be obtained. The set of rules is sometimes termed a code.

Unlawful conduct may include *malfeasance*, an action illegal in and of itself; *misfeasance*, a legal action illegally performed; and *nonfeasance*, a legally mandated action not performed. Remedies include both civil and criminal sanctions, where the criminal sanctions are applied in those cases where the conduct damages the polity itself in some way.

The Evolution of Law

Primitive legal systems were based on *Fas*, religious ritual, and *Mos*, social custom [23, 22]. The law is divined by priests, decreed by a king (acting in a sacred role) or decided by a council of village elders, in all cases acting without the constraints of precedent or a written legal code.

An important step in the advancement of a civilization is a published legal code. Although the source of the law may initially remain unchanged, the act of writing the code makes the law rational rather than mysterious. Many of the initial written codes were still sacred, but later codes were the product of more secular rulers and commissions.

Secular law is based on *Ius*, according to Ulpian, a noted Roman jurist, who defined it as "*the art of goodness and fairness*" [45, Book I] – not exactly the stuff of canon and taboo. Secular law, by dint of its rationality and written form, is inherently changeable because notions of goodness and fairness evolve over time, and what one man can write, another can rewrite.

Changeable, however, does not imply that legal codes and constitutions are either protean or desultory. The law, like all prominent social institutions, is fundamentally conservative. Changes, especially in form, tend to be infrequent and significant, an idea that will serve as the basis for the analysis of the properties of contract.

The Evolution of Contract

In early societies, promises were enforced if they were sacred, either religious obligations or promises made using sacred words. These promises were usually enforced by priests using religious sanctions. One of the original forms of contract in Roman law used the word *spondeo*, which meant to solemnly promise or swear. It was not used outside a sacred context.

Social customs also affect how promises are enforced in legal systems, although the influence is less pronounced than that of religion. Another form of Roman contract was

locatio conductio or hire. Hire was used by tradesmen and laborers, but not by teachers or lawyers because Romans traditionally frowned on professionals selling their services in such common terms. Instead, professional services were typically engaged with the *mandatum* contract, a gratuitous contract which could, however, include a small "gratuity".

But religion and custom play a comparatively small role in the development of contract law because in most cultures, the period of the primacy of *Fas* and *Mos* in the legal system precedes the development of a modern economy. Another factor is that religion and custom are not ordinarily concerned with economic matters, and so obligations in religious and customary legal codes pertain more to delict than contract. (An important exception is marriage.) Thus contract law tends to be secular, derived more from considerations of fairness or economic expediency than from religion and custom.

2.2.3 Rome

The concept of enforceable promises is found in almost every ancient civilization, especially those near the eastern Mediterranean Sea in Egypt, Mesopotamia, Anatolia and the Levant [46, 42].

It was in Rome, however, that contract law was most fully developed. In other civilizations, contract law was limited mostly to several specific transactions such as marriage (actually betrothal) and simple sale.[1] In Rome, it eventually encompassed every form of social and business interaction, including hire of a thing or person, agency, loan, deposit and business partnerships, as well as sale and marriage.

The Roman Constitution

The Rome of legend was founded in 753 B.C. by two brothers, Romulus and Remus, suckled by a wolf. A more likely origin was as a confederacy of three groups from central Italy, the Latins, Sabines and Etruscans. Early Rome was governed by a succession of

[1] As far as we know; ancient history is subject to the accidents of archaeology - it is the *surviving* artifacts which write the history.

kings whose rule was terminated (abruptly, perhaps) in the late sixth century B.C.. A republic was instituted, consisting of several magistrates, four assemblies that met for various purposes, and the Senate [25, 16].

The Magistrates

The king was replaced by two magistrates, the *consuls*, who shared the office. Other subordinate magistracies were created as Rome grew. Most important among these for the law were the two *praetors* who administered the civil law, the *praetor urbanus* and the *praetor peregrinus*. Originally, the former had jurisdiction in cases between Roman citizens and the latter in cases where one or more of the parties were non-citizens. Also noteworthy were the *tribunes* of the plebians[2], the *censors*, who also supervised public morality and the *aediles*, who regulated the marketplaces.

The power of the magistrates was limited by several factors. Besides the division of labor among the different offices, the most important were the shortness of their terms – usually just a year – and the sharing of the office – magistrates could veto each other's actions.

The Assemblies

The four assemblies were the *comitia curiata*, the *comitia centuriata*, the *comitia tributa* and the *concilium plebis*. In each case, the assembly could only be convened by a magistrate who would introduce all legislation, and voting was per block not per individual. The assemblies all had different membership criteria; their legislative domains are less well understood.

comitia curiata

The *comitia curiata* originated during the kingdom. Its composition was based on membership in a curia, an ancient Roman organization with a tribal (fam-

[2]The early Romans were divided into two social classes, the patricians and the plebians; the original distinction may have been based on family or wealth.

ily) and religious basis. During the Republic, it had a largely ceremonial role, mostly validating wills and adoptions.

comitia centuriata

The *comitia centuriata* also may have originated during the kingdom. Its composition was based on membership in a century which, although originally of military significance, came to be based on class and wealth. The highest class (*equites*) and the wealthiest group controlled 98 of the 193 votes.[3] It elected the consuls and praetors and was responsible for acts of war and other major constitutional issues. Because of its importance, the *comitia centuriata* could only be convened by consuls.

comitia tributa

The composition of the *comitia tributa* was also based on membership in a tribe, but with a geographical basis, at least initially. Each Roman citizen belonged to one of thirty-five tribes. The *comitia tributa* had a minor role, electing the minor magistrates, but interestingly its membership criterion was adopted in part for the upper class centuries in the *comitia centuriata*. The *comitia tributa* could be convened by either consuls or praetors.

concilium plebis

The composition of the *concilium plebis* was based on the geographical tribes as well, but membership was restricted to plebians. The distinction between the two classes eventually faded, and by 287 B.C., legislation passed by the *concilium plebis* became binding on all citizens. It elected the plebian tribunes and became the most important legislative body in Rome. This was due both to the increasing proportion of citizens that were plebians, and to the fact that magistrates which convened the other bodies were busier and less numerous than the tribunes.

[3]By comparison, the engineers and musicians each had only two votes.

The Senate

The Senate had its origin as a small, advisory council to the kings. During the Republic, its size and importance grew. Although its decisions, *senatusconsulta*, were not binding during the Republic, its influence was nonpareil; its membership consisted of past consuls, who served for life in the senate after their term of office.

Over the next four centuries, Rome conquered much of the Mediterranean basin, but the republic designed for governing a small, homogeneous community did not survive the strain of ruling a large empire. A succession of emperors displaced much of the functions of the assemblies and the magistrates, and, more gradually, the Senate as well. The Empire from 27 B.C. to A.D. 284 is known as the Principate, and as the Dominate thereafter. Toward the end of the Principate, the Roman Empire split into two, the eastern part governed by Constantinople and the western part still ruled from Rome.

By the early fifth century, Rome itself had succumbed to internal strife and external assaults, but the eastern empire survived intact for several centuries longer. It was an eastern emperor, Justinian, who, during the early sixth century, sponsored the production of the *Corpus Iuris Civilis*, intended as an authoritative collection of the Roman Civil Law. The work was carried out by the foremost legal scholars from Constantinople and Beirut. The *Corpus Iuris Civilis* included the *Codex* (legal statutes), the *Institutiones* (legal textbook) and *Digestum* (legal writings from the great Roman jurists) [31]. The profound influence of Roman law on medieval and modern western civilization is due largely to this work and its survival.

The Civil Law of Rome

Rome produced few great artists and scientists; its public works – roads, aqueducts, buildings – are ruins useful today only for tourism; its military conquests are historical, not geographical, facts; its Civil Law however, still influences daily life in almost every nation of the world.

The historical genesis of Roman law was the XII Tables [44]. The XII Tables represent the Roman law as it existed at the start of the Republic, and so would have been a codification of Roman sacred and customary law. The XII Tables were compiled by a commission of ten men formed around 451 B.C., probably as a result of the struggle between the two social orders. The resulting code was published on ten bronze tablets in the Forum. Two more tablets were added the following year by another commission.

The original tablets were destroyed when the Gauls sacked Rome in the early 4th century, but complete copies existed until at least the first century B.C.. What survives today is fragmentary, but those remnants address most of the subjects of a modern legal code. Included are laws on legal status and capacity, procedure, delict, contract and even zoning regulations: fig and olive trees had to be at least nine feet from the boundary of a property.

During the Republic and the Empire, the three sources of law were statutes (*lex*), magisterial edicts (*edicta*) and juristic opinions (*responsa*) [25].

lex

During the Republic, statutes were legislated by one of the assemblies; during the Principate, it was the *senatusconsulta* which had the force of law; by the Dominate, the main (by far) source of statute law was *constitutiones principes* published by the emperor, a category which included judicial decisions and opinions as well as edicts (the emperor was also a magistrate).

edicta

Edicts were interpretations of the laws published by the higher magistrates at the outset of their term. A magistrate's duties included the administration of disputes within some well-defined sphere, and the edict defined the remedies which would be available to those bringing legal actions before him. The importance of remedies in law is illustrated by the legal maxim *Ubi ius, ibi remedium* – roughly, there can be no right without a remedy.

While the magisterial edicts did not formally alter the statute law, they effectively amended and extended it without the politics and bureaucracy that frequently characterizes the legislative process. Of particular importance here were the edicts of the *praetors*, who had jurisdiction over commercial transactions; much of Roman contract law was derived from their edicts.

responsa

One of the distinguishing characteristics of Roman law was the reliance on the expert opinions of the *jurists*, who were legal scholars rather than judges or lawyers. Perhaps the best modern analogue is the English solicitor, who prepares cases but doesn't argue them. Among the great Roman jurists were Paulus, Ulpian, Modestinus, Papinian and Gaius, all of whom were extensively quoted and cited in Justinian's *Corpus Iuris Civilis*.

The jurists provided advice to both legislators on statute law and magistrates on edicts and also wrote commentaries and textbooks on the law [1]. Their greatest influence on the law, however, was by issuing opinions, termed *responsa*, to the judges trying a case. The *responsa* were similar to the *amicus curiae* briefs filed by modern American lawyers, but were less partisan and tendentious. It was this characteristic, more than any official position the jurist might hold, which gave authority to *responsa*.

Of these three sources, the least influential was *lex*. For example, during the Republic there were only about thirty statutes relating to the private law [25]. The essence of Roman law was in the *edicta* and *responsa*.

Both were practical rather than theoretical: the edicts consisted mainly of specific legal formulae that judges would use in adjudicating cases, while the responses were comments on particular cases. Both were also flexible. The edicts were published on a yearly basis, at the outset of the magistrate's term of office. Although each edict would mostly reprise the past edict, a magistrate could and did issue new interpretations. The jurists

published even more often,[4] and as scholars, were guided but not handcuffed by precedent.

2.2.4 The Contract Law of Rome

Cum nexum faciet mancipiumque,
uti lingua nuncupassit, ita ius esto.[5]
Table VI

This fragment from the XII Tables is the earliest mention of contract in Roman law, and refers to a primitive form known as *nexum*. Linguistically, nexum is a form of the verb *necto*, meaning to bind, tie, join or connect. It appears to denote the relationship between debtor and creditor, but not much is known beyond this [44].

Note the precise correspondence between the obligation according to the law (*ius*) and the spoken words (*lingua*). Also, the use of the verb *nuncupassit*, from *nuncupatio* to pronounce publicly, implies both formality as well as publicity; *nuncupatio* was also used to declare one's heirs, an especially solemn step in Rome. And both of these properties are implicit in *mancipium* (usually known as *mancipatio*), which is coupled in the passage with nexum. *Mancipatio* was the most important form of property conveyance in early Rome.

Its formal origins notwithstanding, Roman contract law was largely pragmatic. Few statutes concern legal obligations; instead, the contract law was largely created by the praetorian edicts and the jurists' responses, both of which were based on real cases, not legal theories. Rome is said to have had a "law of contracts, not contract law" [25, p 165].

The early forms of contract were unilateral in that they concerned the obligations of only one party. They were also *stricti iuris*, which meant that they were construed solely on the letter or form of the contract (*uti lingua nuncupassit*). External issues or rational agreement were irrelevant to the question of enforcement; the obligation existed even if the contract had been induced by threats or fraud.

Later forms of contract were bilateral and were construed in a less literal fashion,

[4] Justinian claimed that the Digest was based on the study of over two thousand distinct legal works
[5] When one performs nexum and mancipium, as he proclaims it, so is the law.

termed *bonae fidei* (in good faith). The judge in a *bonae fidei* case could issue a judgment based on community standards of conduct and intent. Gradually, earlier forms of contract became bilateral and *bonae fidei* as well.

Forms of Roman Contract

aut enim re . . . aut verbis
aut litteris aut consensu.[6]
The Institutes of Gaius, III, § 89

There were four principal forms of Roman contract [25, 43]. This taxonomy was devised by the Roman jurist Gaius in the second century A.D. [1, III, 89], and was retained, verbatim, by the legal scholars who compiled Justinian's *Corpus Iuris Civilis* four centuries later [2, III, 13].

Stipulation (*verbis*)

Stipulation was an oral contract, like nexum, and perhaps as old. It consisted of a promise using a question and answer form. One party would ask, for example, "Do you promise to pay me 100 sesterces?", and the other party would respond, "I promise." The form was rigid to the point that the answer must use the same verb as the question. The earliest verb used in stipulations was *spondere* (to swear solemnly):

Spondesne? *Spondeo.*

and so this contract is also referred to as *sponsio*.

Real (*re*)

The real contracts were created by the delivery of a thing, and were of four types: loan for consumption, loan for use, deposit, and pledge or surety. Because the real contracts were usually gratuitous, they were of little practical importance, except for surety, which usually existed in order to enable a more important transaction.

[6] by the thing itself, by words, by writing, by consent.

Litteral (*litteris*)

Throughout the later Republic and the Principate, the head of the household kept written records of the household accounts. The household account books were of such a nature that their veracity was assumed. The Roman litteral contract was, in essence, a type of entry in the household account books which created an obligation from a real debt or transferred an obligation from one debtor to another.

Consensual (*consensu*)

The consensual contracts were created by agreement, and were of four types: sale, hire, partnership and agency. Sale was the oldest of the consensual contracts, and always involved the transfer of a thing for a price. Hire could be used to rent a thing or to employ people or their services. Partnership was used by the Romans for almost any joint undertaking, short of marriage. It could be used for commercial or charitable purposes, for general or limited areas or activities, and exist for a definite or indefinite term. Agency occurred when one person agreed to perform some service for another.

The consensual contracts are distinguished in both maturity and basis. Other legal systems of the period failed to develop anything comparable in scope or detail, particularly with regard to contracts for sale and hire. And basing obligations on agreement, a subjective state of mind, rather than on a prescribed formal procedure, was an important step in the development of a rational secular law.

2.2.5 Contractual Properties

From the most salient characteristic of the four forms of Roman contract, the following properties are derived:

Obligation ($\Leftarrow$ Stipulation)

Stipulation consisted solely of one party promising to perform some action, and thus

is equivalent to obligation in its simplest form. Obligation requires that a party be bound by the contract, and for bilateral or multilateral contracts, the obligation is mutual.

Consideration ($\Leftarrow$ Real)

In the real contracts, the obligation was created "*by the delivery of the object in respect of which the contract was made*" [7, p 179]. A real contract required a *quid pro quo* – some corporal object to be received by one party in order for there to be a contract. In English law, this concept is termed a *consideration*.

Durability ($\Leftarrow$ Litteral)

The litteral contract was created by an entry in the household account books of the creditor. Although the other forms of contract could be and often were written, particularly after the early Republic, the litteral contract was the only one that, in all cases, had to be written. And while the act of adding an entry to the household account books may have lent credence to the assertion of an obligation, it was the durability of those records which gave them evidentiary value.

Agreement ($\Leftarrow$ Consensual)

Both nominally and in fact, the consensual contracts were based explicitly on agreement. For the stipulation and litteral contracts, the consent of the promiser or debtor was, at best, only minimally required, and even the real contracts could create obligations without assent.

It may seem that deriving the contractual properties from the most salient characteristic of the four forms of Roman contract is somewhat contrived or even a *non sequitor*. Two things, however, argue for this approach.

First, the taxonomy of Roman contracts was devised by an eminent legal scholar, Gaius, who was working at a time, during the early Principate, when both Roman law and legal scholarship were already mature. Four centuries later, this taxonomy was deemed

cogent enough to be included verbatim in Justinian's *Corpus Iuris Civilis*. Thus both the provenance and the endurance of this taxonomy argue for its importance.

Second, it is always easier to modify an existing form than to devise an entirely new form. Thus the mere existence of a new form argues for the significance of the main attribute that differentiates it from other forms.

For example, consider the three main Republican assemblies. Each nominally included the same population[7] but with a different organization for voting and thus a different power structure. The *comitia centuriata* was timocratic,[8] the *comitia tributa* was democratic in the original sense[9] and the *concilium plebis* was democratic in the modern sense. Since the transition from one power basis to another is usually marked by civil war, these differences are significant.

Or, in an area perhaps more familiar to the readers of this work, consider modern computer operating systems. Like the Republican assemblies in Rome, they all do the same thing. But aside from systems whose *raison d'tre* was their proprietary nature (or lack thereof), differences in form were invariably significant. Multics was more secure, Unix was simpler, VM had virtual machines, MVS was compatible, VOS was fault tolerant, et cetera.

Thus, the importance of Gaius's taxonomy and the significance of the distinguishing characteristic of a new form argue that these four properties are fundamental to contract.

2.2.6 Database Properties Comparison

The ACID properties are well-known in database systems. A database transaction processing system (TPS) consists of a set of variables representing stored state, and a set of processes (transactions) that view and update that state. A transaction either succeeds or fails. Each transaction has the following properties [14, p. 6]:

[7] The *concilium plebis* excluded patricians, but in the late Republic, this was an increasingly small minority.
[8] Rule of the propertied class.
[9] A *deme* was a territorial unit in classical Greece.

atomicity Either all or none of a transaction's updates happen.

consistency A transaction's updates do not violate any data integrity constraints.

isolation Transactions appear to each other to execute sequentially.

durability All of a successful transaction's updates survive a system failure.

At first glance, it seemed reasonable to use these properties as a model for the basic properties of electronic commerce transactions. This was not successful, however, partly because of the difference in domains (data versus commodities, services and financial instruments), but also because of shortcomings in the ACID properties themselves.

Consistency is problematic because of its real-world impracticality. To support consistency, a TPS would have to provide minimally for:

1. the definition of state consistency constraints in a formal language that enabled these constraints to be themselves validated for consistency.

2. the testing of each transaction to guarantee that the defined state consistency constraints are not violated.

The impracticality of consistency is strongly argued by the absence of these features in major production transaction processing systems.

Atomicity is problematic because it is implied by consistency and is thus redundant. This can be shown using the following TPS model:

- a set Θ of transactions, $\Theta = \{\theta_i, 0 \leq i\}$;

- a set Δ of variables, $\Delta = \{\delta_j, 0 \leq j\}$; each variable is a natural number and is assumed to have an initial value of 0.

- an execution ξ which consists of a set of tuples, $\langle i, j, k, l \rangle$, $\theta_i \in \Theta$, $\delta_j \in \Delta$, $k \in \mathbb{N}$, $l \in \{\rho, \omega\}$; a tuple denotes a read ($l = \rho$) or a write ($l = \omega$) operation by transaction θ_i on datum δ_j where the value read or written is k.

- a function τ, which maps each tuple to a time value;

$$\tau : \xi \mapsto \mathbb{N}$$

constrained by:

$$\tau\cdot\langle i,j,k,l\rangle = \tau\cdot\langle w,x,y,z\rangle \wedge i = w \Longrightarrow j = x \wedge k = y \wedge l = z$$
$$\tau\cdot\langle i,j,k,l\rangle = \tau\cdot\langle w,x,y,z\rangle \wedge j = x \Longrightarrow i = x \wedge k = y \wedge l = z$$

One transaction is *visible* to another if the second transaction reads a variable after the first one writes one. Using $\dashrightarrow$ to denote visibility:

$$\theta_i \dashrightarrow \theta_j \iff \exists\langle i,x,y,\omega\rangle, \langle j,z,w,\rho\rangle \in \xi \wedge \tau\cdot\langle i,x,y,\omega\rangle < \tau\cdot\langle j,z,q,\rho\rangle \tag{2.1}$$

The notation $\theta_i \dashrightarrow_x \theta_j$ denotes that transaction θ_j is able to read the value of datum δ_x as written by transaction θ_i.

One transaction is *apparent*[10] to another if the second transaction reads the value of a variable written by the first one, either directly or indirectly. Using $\rightarrow$ to denote a direct read:

$$\begin{aligned}\theta_i \rightarrow \theta_j \iff & \exists\langle i,x,y,\omega\rangle, \langle j,x,y,\rho\rangle \in \xi \wedge \tau\cdot\langle i,x,y,\omega\rangle < \tau\cdot\langle j,x,y,\rho\rangle \\ \wedge \nexists & \langle k,x,y,\omega\rangle \in \xi \ni \tau\cdot\langle i,x,y,\omega\rangle < \tau\cdot\langle k,x,y,\omega\rangle \wedge \tau\cdot\langle k,x,y,\omega\rangle < \tau\cdot\langle j,x,y,\rho\rangle\end{aligned} \tag{2.2}$$

The notation $\theta_i \rightarrow_x \theta_j$ denotes that transaction θ_j read the value of datum δ_x written by transaction θ_i.

The operator $\rightsquigarrow$ denotes the transitive closure of $\rightarrow$:

$$\theta_i \rightsquigarrow \theta_n \iff \exists\theta_j,\ldots\theta_m \in \Theta \ni \theta_i \rightarrow \theta_j \rightarrow \ldots\theta_m \rightarrow \theta_n \tag{2.3}$$

The $\rightsquigarrow$ operator defines a binary relation, Γ, on the set of transactions:

$$\Gamma = \Theta \times \Theta \mid \theta_i \rightsquigarrow \theta_j$$

This relation is transitive and reflexive.

[10]The difference between *visible* and *apparent* is one of potential versus actual.

Property 1

Database Isolation

Each transaction appears to each other transaction to execute either before or afterwards (or not at all). To do this requires an additional constraint on the τ *function:*

$$\tau : \xi \mapsto \mathbb{N}$$

further constrained by:

$$\begin{aligned} \theta_i \rightsquigarrow \theta_j \implies \forall a,b,c,d,p,q,r,s, \\ \tau\cdot\langle i,a,b,\omega\rangle < \tau\cdot\langle j,c,d,\rho\rangle \wedge \tau\cdot\langle i,p,q,\rho\rangle < \tau\cdot\langle j,r,s,\omega\rangle \end{aligned} \tag{2.4}$$

What isolation does, in effect, is to make the Γ relation asymmetric.

$$\langle \theta_i, \theta_j \rangle \in \Gamma \implies \langle \theta_j, \theta_i \rangle \notin \Gamma$$

Property 2

Database Atomicity

Either all or none of a transaction's write tuples are visible to other transactions. This imposes one last constraint on the τ *function:*

$$\tau : \xi \mapsto \mathbb{N}$$

further constrained by:

$$\begin{aligned} \langle i,a,b,\omega\rangle, \langle i,c,d,\omega\rangle \in \xi \wedge \theta_i \rightarrow_a \theta_j \implies \\ \langle k,x,y,\rho\rangle \in \xi, k \neq i, \wedge \tau\cdot\langle i,c,d,\omega\rangle < \tau\cdot\langle k,x,y,\rho\rangle \end{aligned} \tag{2.5}$$

Theorem 1

Isolation Implies Atomicity

Show:

$$(2.4) \implies (2.5)$$

Proof:

It is necessary to show that assuming (2.4) *and the antecedent of* (2.5) *implies the consequent of* (2.5)*. The latter conjunct of the antecedent of* (2.5) *states that* θ_j *reads variable* δ_a *from* θ_i*, which means that* $\theta_i \rightsquigarrow \theta_j$*, the antecedent of* (2.4)*. This makes the consequent of* (2.4) *true, and so all write tuples of* θ_i *precede all read tuples of* θ_j*.*

Thus there exists a read tuple ($\langle j, a, b, \rho \rangle$*) which occurs after the other write tuple of* θ_i *(*$\langle i, c, d, \omega \rangle$*) from a different transaction (*$j \neq i$*); this is the consequent of* (2.5)*, which proves the theorem.*

□

From the foregoing it can be seen that the redundancy of atomicity is due to its definition. Atomicity is generally relative to some point of view, and always so in software systems. The definition of database atomicity omits point of view; once it is added, the redundancy becomes clear.

The inclusion of atomicity appears to be due to a mistake in analysis. In [14, p. 7] the origin of atomicity is traced, in part, to the mutual ”I do” promises in the Christian wedding ceremony. But the question and answer part of the ceremony appears, from the repetition of the verb (“Do you . . . ?” “I do.”), to be derived from *stipulatio*. Although stipulation wasn’t used for marriage, it was used for the dowry contract in a form called *dictio dotis*[16].

And so what is represented here is not atomicity or even agreement, but instead is obligation. The bilateral promises in today’s wedding ceremony make the obligation mutual, and it is this mutuality which was mistaken for atomicity.

The other two database properties, however, are analogous to our contractual properties. Durability is essentially the same in both, while database isolation is similar to obligation, in that all changes to the status quo are given effect simultaneously.

2.3 Security

Reporter: “Why do you rob banks?”

Willie Sutton[11]: “Because that’s where the money is.”

Security, too, has an economic rationale. Contracts are legally enforceable promises, but those promises are about the transfer of goods, services and financial instruments – objects of economic value.

2.3.1 Computer Security

Security in computer systems is concerned with the protection of data against willful threats to its integrity, privacy and availability, either while the data is stored in the system or during its transmission between system or user processes [28, 9]. These two facets of security are termed here storage and communication security, respectively.

In both cases, data is protected by the system reference monitor, which consists of the hardware and software mechanisms which arbitrate all process data accesses, either preventing or auditing unauthorized accesses. The reference monitor arbitrates data accesses based on various security-relevant criteria which are used to define the system security policy. The types of security criteria supported on the system along with the language used to define the security policy constitute the system security policy model [37].

Storage Security

For stored data, typical reference monitor hardware mechanisms include separate process address spaces and read-write memory access controls. The reference monitor uses the former to control which data a process can access, and the latter to control which data a process can alter. Privileged instructions and a supervisor state bit may also be used to control access at the hardware layer.

[11] American bank robber and monetary theorist.

Above the hardware layer are software mechanisms which directly enforce the storage security policy. These mechanisms consist of access control and audit data structures which the reference monitor uses to determine whether to grant and/or record process data access requests. The access data structures may be lists or sets processed according to a specified algorithm, or they may be statements in a computer or logical language.

Standard storage security criteria include user or process characteristics, data characteristics or values, system status, and type of access. The types of access include writing, reading, and locking for exclusive access, which correspond to the security goals of integrity, privacy and availability. Additional security criteria may include system load [47], access time or locale, or other factors that the system can reliably compute [37].

Access data structures may be defined on an object, object container, object class, subject or system basis, or any combination of these. These structures are usually defined by system administrators, programmers or data object owners.

Communication Security

For transmitted data, the reference monitor usually has no special hardware mechanisms. There may be multiple transmission paths between the sender and receiver, which would increase the probability of data availability, and in some systems the transmission media may be encased in a tamper-resistant or tamper-evident envelope to provide protection against or auditing of line tapping.

The main software mechanism in communication security is cryptography [29, 17, 18], which is used primarily to support privacy and integrity, and secondarily to support origin and uniqueness.

Privacy ensures that only the sender and receiver of a message can view the message data; integrity ensures that the receiver of a message can verify the genuineness of the message data; origin ensures that the receiver of a message can verify the message sender's identity; uniqueness ensures that the receiver of a message can verify the freshness of the

data. Origin may be verifiable by additional parties other than the receiver; this is termed non-repudiation.

Cryptography is used in security protocols to support some or all of these properties for a given message according to the communication security policy. The criteria and languages used to define this policy are normally simpler than those used to define storage policy.

2.3.2 Security Properties

The security properties for electronic commerce are derived directly from the above communication security properties:

privacy

Transaction data can only be viewed by the parties; if the transaction has more than two participants, each participant can only view data it either sent or received.

origin

The identity of the sender of transaction data shall be verifiable by the receiver and, if the data represents an obligation, origin shall be non-repudiable.

integrity

The genuineness of transaction data shall be verifiable by the receiver and, if the data represents an obligation, by independent arbiters.

uniqueness

The newness of data can be verified by the receiver.

2.4 Structure

Structural properties define the relationships among elements of a design. In commerce, the main structural issue is the number and kind of participants permitted in a transaction. This

is termed transaction scalability.

In computer systems, scalability refers to the ability of a system to respond to increased performance demands in a linear fashion. In a database system, for example, an increase in the number of transactions or a change in the type of database operations should not require any changes in the existing system hardware and software beyond the addition of hardware – processors, disk controllers – to handle the new performance demands.

Commercial transactions range from the simple, involving a customer purchasing one or two items from a merchant using cash, to the complex, involving many participants in different roles using arbitrary exchange media. Even simple transactions are not as simple at they first appear – the purchase usually includes taxation and some type of credit verification, and may also require licensing for goods under government control or reservations for goods that are not to be transferred immediately. And for any transaction which could be disputed, an impartial record will be required to be used in arbitration.

Accordingly, two kinds of transaction scalability are important in commerce. The first is *quantitative* - the number of transaction participants may vary considerably. The second is *qualitative* - the role of transaction participants will also vary considerably; some may be direct participants, actively involved in the exchange as buyers or sellers, while others may be indirect participants, passively involved in the exchange as licensors or registrars or taxers.

2.5 Summary

In this chapter, contract was argued to be fundamental to a modern economy, and thus governments have a valid reason for enforcing legal promises. Next, the four properties of contract were derived using Roman law as their basis. Each property corresponds directly to the predominant characteristic of one of the four forms of Roman contract. Rome was selected as a reference because its legal system, particularly the Roman law of contracts, was the basis for a commercial empire that lasted a thousand years.

The properties of database transactions were compared to the contractual properties of commercial transactions. Two of the database properties, isolation and durability, were found to correspond to the contractual properties of obligation and durability. But the other two database properties, atomicity and consistency, were shown to be redundant or impractical.

Security was also argued to have an economic rationale. For security, the goal is to prevent or minimize crime rather than to promote economic growth. The use of computer networks for electronic commerce adds opportunity to motive. A simple model was presented for secure storage and communication, focusing on communication security because of the use of communication protocols to provide distributed services in electronic commerce. The security properties for electronic commerce were taken directly from the basic network security properties of privacy, integrity, origin and uniqueness.

Finally, structural properties were addressed. Since commercial transactions can have multiple participants in different roles, electronic commercial transactions should be able to include multiple processes without constraining the process types. These two properties were defined as quantitative and qualitative scalability.

Chapter 3

Order

3.1 Overview

order *n.* [24]

1. A condition of logical or comprehensible arrangement among the separate elements of a group.

Ideas of order encompass the physical, the mathematical, the etiological, the chronological, the musical, even the poetic:

Oh! Blessed rage for order, pale Ramon,
The maker's rage to order words of the sea,
Words of the fragrant portals, dimly-starred,
And of ourselves and of our origins,
In ghostlier demarcations, keener sounds.

The Idea of Order at Key West
Wallace Stevens

Here the more mundane focus is on commerce. The order of events in commerce can be used to establish the course of a business negotiation, to define relationships among the items being exchanged in a transaction, or to validate that an event occurred within a specified period. The first two of these are concerned with causal order, while the last is based on temporal order. Accordingly, in electronic commerce it will be necessary to determine the causal and temporal order among some of the events in a transaction.

Electronic transactions are conducted by means of protocols, which are used for communication among processes in distributed systems. A protocol is a set of messages along with defined permissible sequences of those messages, and so the events in an electronic transaction correspond to the transmission and reception of messages in an electronic commerce protocol.

Order protocols have long been of interest in distributed systems research and practice [20, 5], where the computed or induced order of events can be used to establish atom-

icity, queuing priority, data consistency and other important computational or systematic properties.

The major issues for order protocols are correctness, consistency, completeness and efficiency. Correctness and consistency are essential to any protocol and will be the focus of this chapter. Completeness may not be possible or practical in all cases, and it may not be required for some purposes. Efficiency is always a concern in communication systems, but is a matter of degree as long as the network is not congested.

The correct, consistent computation of order is discussed in the next section; existing order protocols are described and analyzed in section 3.3; lastly, clio, an order protocol designed for electronic commerce, is defined and analyzed in section 3.4.

3.2 Correct, Consistent Order

In electronic commerce the concern is with computing, not inducing, order. Protocols that compute order do so in many ways. The choice of method depends on which events are to be ordered, how (causally or temporally) they are to be ordered, efficiency, trust considerations, and various other factors.

All such protocols, however, effectively compute a binary relation on the set of system events, where this relation is a total or partial order. A binary relation is a total order if all events are ordered: for all events x and y, either $\langle x,y\rangle$ or $\langle y,x\rangle$ is in the relation. Otherwise the relation is a partial order. In a partial order, if neither an event tuple nor its negation are in the relation, the two events in the tuple are said to be *simultaneous*. Intuitively, if $\langle x,y\rangle$ is a tuple in an order relation, then event x *preceded* event y in some causal or temporal sense.

An order relation is computed by first associating with each event an **ordinal**, which is data intended to connote or imply order. If the ordinal is numeric, the order relation may be computed using a comparison operator (usually $<$). Or if the ordinal is a set, the order relation may be computed using a set inclusion operator (usually $\in$). The ordinal data type

and the numeric or set relational operators are protocol specific. The association between event and ordinal need not be immediate or complete, depending on the purpose of the protocol.

The correctness and consistency of protocols can be analyzed by formalizing this computation. The next section defines a formal model for computing order, and the following section discusses temporal and causal order and defines reference protocols for computing both types of order. These reference protocols will be used to analyze existing order protocols.

3.2.1 System Model

Definitions

Definition 1

Event

An *event*, denoted $e_i,\ 0 \leq i$, is a computation.

Definition 2

System

A *system*, denoted $\mathcal{S}$, is a set of events:

$$\mathcal{S} = \{e_i\}$$

Functions and Properties

Function 1

Order Protocol

An *order protocol*, denoted $\alpha, \beta, \ldots\ \omega$, from Π, the set of order protocols, is a function from a system to a binary relation on that system. The **order relation** computed by α on system $\mathcal{S}$ is denoted $\alpha{\cdot}\mathcal{S}$.

$$\alpha : \{\mathcal{S}\} \mapsto \mathcal{S} \times \mathcal{S}$$

constrained by:

$$\forall \alpha \in \Pi,\ \forall e_i \in \mathcal{S},\ \langle e_i, e_i \rangle \notin \alpha \cdot \mathcal{S} \tag{3.1}$$

$$\forall \alpha \in \Pi,\ \forall e_i, e_j \in \mathcal{S},\ \langle e_i, e_j \rangle \in \alpha \cdot \mathcal{S} \implies \langle e_j, e_i \rangle \notin \alpha \cdot \mathcal{S} \tag{3.2}$$

$$\forall \alpha \in \Pi,\ \forall e_i, e_j, e_k \in \mathcal{S},$$
$$\langle e_i, e_j \rangle \in \alpha \cdot \mathcal{S} \wedge \langle e_j, e_k \rangle \in \alpha \cdot \mathcal{S} \implies \langle e_i, e_k \rangle \in \alpha \cdot \mathcal{S} \tag{3.3}$$

Function 2

`time()`

`time` is a function from the set of events to the set of natural numbers:

$$\texttt{time} : \mathcal{S} \mapsto \mathbb{N}.$$

constrained by:

$$\forall e_i, e_j \in \mathcal{S},\ \texttt{time}(e_i) \leq \texttt{time}(e_j) \oplus \texttt{time}(e_j) \leq \texttt{time}(e_i) \tag{3.4}$$

$$\forall e_i, e_j, e_k \in \mathcal{S},\ \texttt{time}(e_i) < \texttt{time}(e_j) \wedge \texttt{time}(e_j) < \texttt{time}(e_k)$$
$$\implies \texttt{time}(e_i) < \texttt{time}(e_k) \tag{3.5}$$

The `time` function is strictly isotonic with respect to the rate of events. That is, each event takes at least one “clock tick”.

Function 3

`prec()`

`prec` is a function from the set of events to a set of those events:

$$\texttt{prec} : \mathcal{S} \mapsto \mathcal{X}$$

constrained by:

$$\mathcal{X} \in 2^{\mathcal{S}} \wedge \ |\mathcal{X}| \leq 2 \tag{3.6}$$

$$\forall e_i \in \mathcal{S},\ e_i \notin \texttt{prec}(e_i) \tag{3.7}$$

$$\forall e_i, e_j \in \mathcal{S},\ e_i \in \texttt{prec}(e_j) \implies e_j \notin \texttt{prec}(e_i) \tag{3.8}$$

$$\forall e_i, e_j, e_k \in \mathcal{S},\ e_i \in \texttt{prec}(e_j) \wedge e_i \in \texttt{prec}(e_k) \implies j = k \tag{3.9}$$

This set is termed the **preceding event set** of the event in question. The transitive closure of `prec` is `prec*`:

$$\texttt{prec*} : \mathcal{S} \mapsto 2^{\mathcal{S}}$$

constrained by:

$$\forall e_i, e_j, e_k \in \mathcal{S},\ e_i \in \texttt{prec*}(e_j) \wedge e_j \in \texttt{prec*}(e_k) \implies e_i \in \texttt{prec*}(e_k) \tag{3.10}$$

Note that irreflexivity and asymmetry also hold for `prec*`.

Function 4

$\Theta()$

Θ is a protocol specific function. Its range is the set of events, but the domain is protocol specific:

$$\Theta_\alpha : \mathcal{S} \mapsto \mathbb{D}_\alpha$$

The Θ function denotes the event ordinal.

Property 3

`prec`-`time` correlation

Because of the isotonicity of the `time` *function, the functions* `prec` *and* `prec*`*, and the function* `time` *are correlated:*

$$\forall e_i, e_j \in \mathcal{S},\ e_i \in \texttt{prec}(e_j) \implies \texttt{time}(e_i) < \texttt{time}(e_j)$$

$$\forall e_i, e_j \in \mathcal{S},\ e_i \in \texttt{prec*}(e_j) \implies \texttt{time}(e_i) < \texttt{time}(e_j)$$

Order Protocol Consistency

Property 4

Order Protocol Consistency

Protocols α *and* β *are* consistent *iff* $\alpha \cdot \mathcal{S}$ *does not contain the negation of any tuple in* $\beta \cdot \mathcal{S}$.

Using the symbol $\|$ *to denote order consistency:*

$$\alpha \parallel \beta \iff \langle e_i, e_j \rangle \in \alpha \cdot \mathcal{S} \implies \langle e_j, e_i \rangle \notin \beta \cdot \mathcal{S}$$

– or equivalently –

$$\alpha \nparallel \beta \iff \exists \langle e_i, e_j \rangle \in \alpha \cdot \mathcal{S} \ni \langle e_j, e_i \rangle \in \beta \cdot \mathcal{S}$$

Theorem 2

Protocol Consistency Symmetry

Show:

$$\alpha \parallel \beta \iff \beta \parallel \alpha$$

Proof: *by cases*

1. Show:

$$\alpha \parallel \beta \implies \beta \parallel \alpha$$

Subproof: *by contradiction*

(a) $\alpha \parallel \beta$

Hypothesis.

(b) $\beta \nparallel \alpha$

Contradictory Hypothesis.

(c) $\langle e_i, e_j \rangle \in \alpha \cdot \mathcal{S} \implies \langle e_j, e_i \rangle \notin \beta \cdot \mathcal{S}$

Property 4, Step 1a, Modus Ponens.

(d) $\exists \langle e_j, e_i \rangle \in \beta \cdot \mathcal{S} \ni \langle e_i, e_j \rangle \in \alpha \cdot \mathcal{S}$

Property 4, Step 1b, Modus Ponens.

(e) $\beta \parallel \alpha$

Step 1c, Step 1d, Contradiction.

(f) $\alpha \parallel \beta \implies \beta \parallel \alpha$

Step 1a, Step 1e, Deduction.

□

2. Show:

$$\beta \parallel \alpha \implies \alpha \parallel \beta$$

Subproof: *by contradiction*

$\therefore$ *Similarly.*

□

3. $\alpha \parallel \beta \iff \beta \parallel \alpha$

Step 1, Step 2, Tautology.

□

Order protocol consistency is also reflexive, since by definition every relation is consistent with itself, but order protocol consistency is not transitive. Consider two order protocols α and β, which compute $\alpha \cdot \mathcal{S}$ and $\beta \cdot \mathcal{S}$, each containing one tuple, $\langle e_i, e_j \rangle$ and $\langle e_j, e_i \rangle$, respectively. These protocols are not consistent because their computed order relations contain contradictory tuples, but both would be consistent with a third order protocol γ whose order relation contained no tuples. Thus:

$$\alpha \parallel \gamma \wedge \gamma \parallel \beta \wedge \alpha \nparallel \beta$$

Order Protocol Correctness

Property 5

Order Protocol Correctness

Order protocol α *is* correct *vis-á-vis order protocol* β *iff* $\beta{\cdot}\mathcal{S}$ *contains all tuples in* $\alpha{\cdot}\mathcal{S}$.

Using the symbol $\subseteq$ *to denote correctness:*

$$\alpha \subseteq \beta \iff \langle e_i, e_j \rangle \in \alpha{\cdot}\mathcal{S} \implies \langle e_i, e_j \rangle \in \beta{\cdot}\mathcal{S}$$

– or equivalently –

$$\alpha \not\subseteq \beta \iff \exists e_i, e_j \in \mathcal{S} \ni \langle e_i, e_j \rangle \in \alpha{\cdot}\mathcal{S} \wedge \langle e_i, e_j \rangle \notin \beta{\cdot}\mathcal{S}$$

Theorem 3

Protocol Correctness Implies Consistency

Show:

$$\alpha \subseteq \beta \implies \alpha \parallel \beta$$

Proof:

1. $\alpha \subseteq \beta$

   Hypothesis.

2. $\langle e_i, e_j \rangle \in \alpha{\cdot}\mathcal{S} \implies \langle e_i, e_j \rangle \in \beta{\cdot}\mathcal{S}$

   Property 5, Step 1, Modus Ponens.

3. $\langle e_i, e_j \rangle \in \beta{\cdot}\mathcal{S} \implies \langle e_j, e_i \rangle \notin \beta{\cdot}\mathcal{S}$

   (3.2), Step 2, Modus Ponens.

4. $\langle e_i, e_j \rangle \in \alpha{\cdot}\mathcal{S} \implies \langle e_j, e_i \rangle \notin \beta{\cdot}\mathcal{S}$

   Step 2, Step 3, Transitivity of $\implies$.

5. $\alpha{\cdot}\mathcal{S} \parallel \beta{\cdot}\mathcal{S}$

   Property 4, Step 4, Modus Ponens.

6. $\alpha \cdot \mathcal{S} \subseteq \beta \cdot \mathcal{S} \implies \alpha \cdot \mathcal{S} \parallel \beta \cdot \mathcal{S}$

   Step 1, Step 5 Deduction.

□

Theorem 4

Protocol Correctness Transitivity

Show:

$$\alpha \subseteq \beta \wedge \beta \subseteq \gamma \implies \alpha \subseteq \gamma$$

Proof: *by contradiction*

1. $\alpha \subseteq \beta \wedge \beta \subseteq \gamma$

   Hypothesis.

2. $\alpha \not\subseteq \gamma$

   Contradictory Hypothesis.

3. $\langle e_i, e_j \rangle \in \alpha \cdot \mathcal{S} \implies \langle e_i, e_j \rangle \in \beta \cdot \mathcal{S}$

   Property 5, Step 1, Modus Ponens.

4. $\langle e_i, e_j \rangle \in \beta \cdot \mathcal{S} \implies \langle e_i, e_j \rangle \in \gamma \cdot \mathcal{S}$

   Property 5, Step 1, Modus Ponens.

5. $\exists e_i, e_j, \langle e_i, e_j \rangle \in \alpha \cdot \mathcal{S} \wedge \langle e_i, e_j \rangle \notin \gamma \cdot \mathcal{S}$

   Property 5, Step 2, Modus Ponens.

6. $\langle e_i, e_j \rangle \in \beta \cdot \mathcal{S}$

   Step 3, Step 5, Modus Ponens.

7. $\langle e_i, e_j \rangle \in \gamma \cdot \mathcal{S}$

   Step 4, Step 6, Modus Ponens.

8. $\alpha \subseteq \gamma$

   Step 5, Step 7, Contradiction.

9. $\alpha \subseteq \beta \wedge \beta \subseteq \gamma \Longrightarrow \alpha \subseteq \gamma$

Step 1, Step 8, Deduction.

□

Order protocol correctness is also reflexive, since every relation contains itself, but is not symmetric because set inclusion is not symmetric.

Note: In [30], *causal consistency* is defined as the inclusion of their reference causal relation in the subject order relation. This is correct, as Theorem 3 confirms, but not in line with standard usage in which consistency simply implies the absence of contradiction. The *order correctness* property is defined as *characterizing causality.*

Property 6

Order Protocol Equivalence

Protocol α *is* equivalent *to protocol* β *if and only if protocols* α *and* β *are both correct vis-á-vis to each other. Using the symbol* $\equiv$ *to denote protocol equivalence:*

$$\alpha \equiv \beta \iff \alpha \subseteq \beta \wedge \beta \subseteq \alpha$$

Order protocol equivalence is an equality relation, and so is reflexive, symmetric and transitive.

3.2.2 Temporal and Causal Order

An order relation may be either temporal or causal. In a temporal order relation, events are ordered according to some analogue of natural time, usually a physical clock. In a causal order relation, events are ordered according to computational dependencies. An event e_z is regarded as being computationally dependent upon another event e_a if either:

- e_a is an earlier event in the same process as e_z;
- or e_a is the sending event and e_z is the receiving event of the same message;

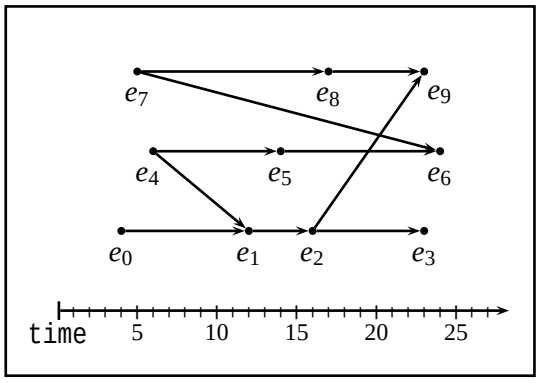

Figure 3.1: System $\mathcal{A}$

- or there is a chain of computationally dependent events $e_b \ldots e_y$ such that:
  - e_b is dependent on e_a;
  - and e_z is dependent on e_y.

The difference between causal and temporal order can be seen by looking at the events in $\mathcal{A}$, a subset of $\mathcal{S}$, depicted in Figure 3.1. The `time` function is shown at the bottom, with each hash mark denoting one unit of time. As noted earlier, an event is denoted e_i, where i is unique.

A computational connection can be visualized if the events in the figure are regarded as vertices in a directed graph, with the arrows as edges. There is a computational connection from one event to another if a path can be traced in the graph from one event to the other using the edges.

A temporal order relation for this system would contain the tuples $\langle e_0, e_6 \rangle$ and $\langle e_7, e_6 \rangle$, because both events e_0 and e_7 happen before event e_6 chronologically. But a causal relation would only contain the later tuple, since there is no computational connection between events e_0 and e_6.

Most order protocols compute causal rather than temporal order, partly because of the difficulty of maintaining an accurate clock over a distributed system, but also because

of the lack of application of temporal order to interesting problems. One temporal order application would be computing precedence or priority by resource servers among competing client requests.

Cronos (χ)

A circumstantial evidence case often rests heavily on chronology.

David Kendall, Counsel to the President

Impeachment Hearings, January 1999

Cronos, denoted χ, is the reference temporal order protocol; it uses the `time` function from the model to compute the event ordinal:

$$\Theta_\chi(e_i) = \texttt{time}(e_i) \tag{3.11}$$

The event ordinals for $\mathcal{A}$ (Figure 3.1) are shown in Table 3.1. The order relation for Cronos is computed by comparing the ordinals for each pair of events:

$$\langle e_i, e_j \rangle \in \chi{\cdot}\mathcal{S} \iff \Theta_\chi(e_i) < \Theta_\chi(e_j) \tag{3.12}$$

The order relation for Cronos computed for $\mathcal{A}$ is shown in Table 3.2. This relation is a partial order since events e_3 and e_9 have the same ordinal and so are simultaneous.

There are, however, inherent problems with temporal order protocols for real systems in which protocols are executed by processes using local clocks on machines connected by computer networks with varying transmission latency and reliability.

clock synchronization

Clock synchronization is both expensive and often impractical for a given purpose; if the local machine clocks are poorly synchronized, then the order of events computed by temporal order protocols would be altered.

clock skew

Even if machine clocks are synchronized, the interval between updates may allow

Event	$\Theta_\chi(e)$
e_0	4
e_1	12
e_2	16
e_3	23
e_4	6
e_5	14
e_6	24
e_7	5
e_8	17
e_9	23

Table 3.1: Cronos Ordinals for $\mathcal{A}$

$\chi \cdot \mathcal{A}$			
$\langle e_0, e_7 \rangle$	$\langle e_0, e_4 \rangle$	$\langle e_0, e_1 \rangle$	$\langle e_0, e_5 \rangle$
$\langle e_0, e_2 \rangle$	$\langle e_0, e_8 \rangle$	$\langle e_0, e_3 \rangle$	$\langle e_0, e_9 \rangle$
$\langle e_0, e_6 \rangle$			
$\langle e_7, e_4 \rangle$	$\langle e_7, e_1 \rangle$	$\langle e_7, e_5 \rangle$	$\langle e_7, e_2 \rangle$
$\langle e_7, e_8 \rangle$	$\langle e_7, e_3 \rangle$	$\langle e_7, e_9 \rangle$	$\langle e_7, e_6 \rangle$
$\langle e_4, e_1 \rangle$	$\langle e_4, e_5 \rangle$	$\langle e_4, e_2 \rangle$	$\langle e_4, e_8 \rangle$
$\langle e_4, e_3 \rangle$	$\langle e_4, e_9 \rangle$	$\langle e_4, e_6 \rangle$	
$\langle e_1, e_5 \rangle$	$\langle e_1, e_2 \rangle$	$\langle e_1, e_8 \rangle$	$\langle e_1, e_3 \rangle$
$\langle e_1, e_9 \rangle$	$\langle e_1, e_6 \rangle$		
$\langle e_5, e_2 \rangle$	$\langle e_5, e_8 \rangle$	$\langle e_5, e_3 \rangle$	$\langle e_5, e_9 \rangle$
$\langle e_5, e_6 \rangle$			
$\langle e_2, e_8 \rangle$	$\langle e_2, e_3 \rangle$	$\langle e_2, e_9 \rangle$	$\langle e_2, e_6 \rangle$
$\langle e_8, e_3 \rangle$	$\langle e_8, e_9 \rangle$	$\langle e_8, e_6 \rangle$	
$\langle e_3, e_6 \rangle$			
$\langle e_9, e_6 \rangle$			

Table 3.2: Cronos Order Relation for $\mathcal{A}$

differences in the local clock speed to alter the computed order of events.

clock imprecision

In some systems there may be a trade-off between clock synchronization and granularity; if the clock granularity is less than the required event granularity, then the event order distinction becomes too coarse.

Telos (τ)

To know truly is to know by causes.

Francis Bacon

Telos, denoted τ, is the reference causal order protocol; it uses the transitive closure of the preceding event set from the model as the event ordinal. Logically, the preceding event set is the set of prior events which could have influenced or caused the present event.

Event	$\Theta_{tau}(e)$
e_0	$\emptyset$
e_4	$\emptyset$
e_7	$\emptyset$
e_1	$\{e_0, e_4\}$
e_5	$\{e_4\}$
e_2	$\{e_0, e_4, e_1\}$
e_8	$\{e_7\}$
e_3	$\{e_0, e_4, e_1, e_2\}$
e_9	$\{e_0, e_4, e_1, e_2, e_7, e_8\}$
e_6	$\{e_7, e_5, e_4\}$

Table 3.3: Telos Ordinals for $\mathcal{A}$

$\tau \cdot \mathcal{A}$			
$\langle e_0, e_1 \rangle$	$\langle e_0, e_2 \rangle$	$\langle e_0, e_3 \rangle$	$\langle e_0, e_9 \rangle$
$\langle e_1, e_2 \rangle$	$\langle e_1, e_3 \rangle$	$\langle e_1, e_9 \rangle$	
$\langle e_2, e_3 \rangle$	$\langle e_2, e_9 \rangle$		
$\langle e_4, e_1 \rangle$	$\langle e_4, e_2 \rangle$	$\langle e_4, e_3 \rangle$	$\langle e_4, e_9 \rangle$
$\langle e_5, e_6 \rangle$			
$\langle e_7, e_8 \rangle$	$\langle e_7, e_9 \rangle$	$\langle e_7, e_6 \rangle$	
$\langle e_8, e_9 \rangle$			

Table 3.4: Telos Order Relation for $\mathcal{A}$

The Telos event ordinal is computed as follows:

$$\Theta_\tau(e_i) = \texttt{prec*}(e_i) \tag{3.13}$$

The event ordinals for $\mathcal{A}$ (Figure 3.1) are shown in Table 3.3. Telos computes the order relation using set inclusion:

$$\langle e_i, e_j \rangle \in \tau \cdot \mathcal{S} \iff e_i \in \Theta_\tau(e_j) \tag{3.14}$$

The order relation for Telos computed for $\mathcal{A}$ is shown in Table 3.4.

Theorem 5

Correctness of Telos vis-à-vis Cronos

Show:

$$\langle e_i, e_j \rangle \in \tau \cdot \mathcal{S} \implies \langle e_i, e_j \rangle \in \chi \cdot \mathcal{S}$$

Proof:

1. $\langle e_i, e_j \rangle \in \tau \cdot \mathcal{S}$

   Hypothesis.

2. $e_i \in \Theta_\tau(e_j)$

(3.14), Step 1, Modus Ponens.

3. $e_i \in \texttt{prec}^*(e_j)$

   (3.13), Step 2, Equality.

4. $\texttt{time}(e_i) < \texttt{time}(e_j)$

   Step 3, Property 3, Modus Ponens.

5. $\Theta_\chi(e_i) < \Theta_\chi(e_j)$

   (3.11), Step 5, Equality.

6. $\langle e_i, e_j \rangle \in \chi \cdot \mathcal{S}$

   (3.12), Step 5, Modus Ponens.

7. $\langle e_i, e_j \rangle \in \tau \cdot \mathcal{S} \implies \langle e_i, e_j \rangle \in \chi \cdot \mathcal{S}$

   Step 1, Step 6, Deduction.

□

Corollary 1

Consistency of Telos and Cronos

Show:

$$\tau \parallel \chi$$

Proof:

$\therefore$ *This follows directly from Theorem 5, Theorem 3 and Modus Ponens.*

□

Note that Telos is intended to serve only as a reference causal protocol; it would be inefficient for long or complex computations since the size of the context is unbounded.

3.3 Existing Protocols

Examples of existing protocols are discussed and analyzed in this section, but first the system model is extended to accommodate some of these protocols.

Definition 3

Process

A *process* is a sequence of events. Each event is associated with one process and each process has a unique identifier.

Function 5

`pid()`

`pid` is a function from the set of events to the set of natural numbers:

$$\texttt{pid} : \mathcal{S} \mapsto \mathbb{N}.$$

$\texttt{pid}(e_i)$ returns the identifier of event e_i's process.

Definition 4

Message

A *message* is a binary relation, $\mathbb{M}$, on the set of events:

$$\langle e_i, e_j \rangle \in \mathbb{M} \iff e_i \in \texttt{prec}(e_j) \wedge \texttt{pid}(e_i) \neq \texttt{pid}(e_j)$$

Message $\langle e_i, e_j \rangle$ can also be denoted m_i. Note that messages are otherwise implicit in the model; this definition serves only to provide a naming scheme because some order protocols associate ordinals with messages, either instead of or in addition to event ordinals.

`Function 3` (*continued*)

`prec()`

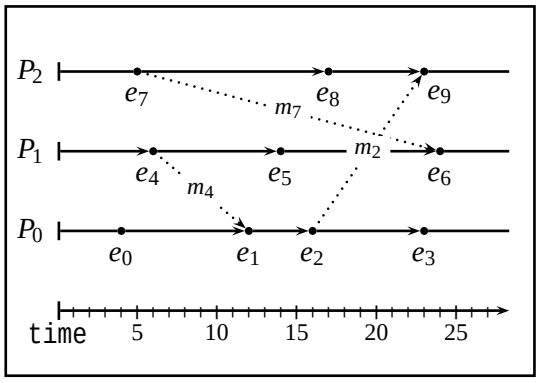

Figure 3.2: System $\mathcal{A}'$

constrained by:

$$\forall e_i, e_j \in \mathcal{S},$$

$$\{e_i, e_j\} = \texttt{prec}(e_k) \implies \texttt{pid}(e_i) = \texttt{pid}(e_k) \oplus \texttt{pid}(e_j) = \texttt{pid}(e_k) \tag{3.15}$$

$$\forall e_i, e_j \in \mathcal{S},\ \texttt{pid}(e_i) = \texttt{pid}(e_j) \implies e_i \in \texttt{prec*}(e_j) \ \oplus\ e_j \in \texttt{prec*}(e_i) \tag{3.16}$$

$$\forall e_i \in \mathcal{S},\ \exists e_j \in \mathcal{S} \ni \texttt{pid}(e_j) = \texttt{pid}(e_i) \wedge e_j \in \texttt{prec}(e_i)$$

$$\vee\ \nexists e_k \in \mathcal{S} \ni \texttt{pid}(e_k) = \texttt{pid}(e_i) \wedge e_k \in \texttt{prec*}(e_i) \tag{3.17}$$

In Figure 3.2, $\mathcal{A}$ has been redrawn as $\mathcal{A}'$ to include processes and messages. The processes are indicated as a sequence of horizontal, solid vectors, while the messages are drawn as diagonal, dotted vectors.

3.3.1 Lamport Clock (λ)

In the Lamport Clock protocol, denoted λ and described in [20], each process maintains a logical clock; its value is initially zero, and it is incremented as each event occurs. This logical clock is used as the ordinal for this protocol:

$$\Theta_\lambda(e_i) := 1 + \texttt{max}\{\Theta_\lambda(e_j), e_j \in \texttt{prec}(e_i)\} \tag{3.18}$$

The event ordinals computed by the Lamport Clock protocol for $\mathcal{A}'$ are shown in Table 3.5.

Event	$\Theta_\lambda(e)$
e_0	1
e_1	2
e_2	3
e_3	4
e_4	1
e_5	2
e_6	3
e_7	1
e_8	2
e_9	4

Table 3.5: LmpClk Ordinals for $\mathcal{A}'$

$\lambda\cdot\mathcal{A}'$			
$\langle e_0,e_1\rangle$	$\langle e_0,e_5\rangle$	$\langle e_0,e_2\rangle$	$\langle e_0,e_8\rangle$
$\langle e_0,e_3\rangle$	$\langle e_0,e_9\rangle$	$\langle e_0,e_6\rangle$	
$\langle e_7,e_1\rangle$	$\langle e_7,e_5\rangle$	$\langle e_7,e_2\rangle$	$\langle e_7,e_8\rangle$
$\langle e_7,e_3\rangle$	$\langle e_7,e_9\rangle$	$\langle e_7,e_6\rangle$	
$\langle e_4,e_1\rangle$	$\langle e_4,e_5\rangle$	$\langle e_4,e_2\rangle$	$\langle e_4,e_8\rangle$
$\langle e_4,e_3\rangle$	$\langle e_4,e_9\rangle$	$\langle e_4,e_6\rangle$	
$\langle e_1,e_2\rangle$	$\langle e_1,e_3\rangle$	$\langle e_1,e_9\rangle$	$\langle e_1,e_6\rangle$
$\langle e_{1,15},e_2\rangle$	$\langle e_{1,15},e_3\rangle$	$\langle e_{1,15},e_9\rangle$	$\langle e_{1,15},e_6\rangle$
$\langle e_8,e_2\rangle$	$\langle e_1,e_3\rangle$	$\langle e_8,e_9\rangle$	$\langle e_1,e_6\rangle$
$\langle e_2,e_3\rangle$	$\langle e_2,e_9\rangle$		
$\langle e_6,e_3\rangle$	$\langle e_6,e_9\rangle$		

Table 3.6: LmpClk Order Relation for $\mathcal{A}'$

Note events e_6 and e_9. The preceding event set for each contains two events, and so their ordinals are based on two preceding events, one in the same process and the other in a different process:

$$\Theta_\lambda(e_6) = 1 + \mathsf{max}(\Theta_\lambda(e_5), \Theta_\lambda(e_7))$$

$$\Theta_\lambda(e_9) = 1 + \mathsf{max}(\Theta_\lambda(e_2), \Theta_\lambda(e_8))$$

For e_6, the preceding event in the same process, e_5, has a greater ordinal than that of the event in the different process, e_7, while for e_9 the opposite is true.

The order relation for Lamport Clock for a system $\mathcal{S}$ is computed by comparing the ordinals for each pair of events:

$$\langle e_i, e_j\rangle \in \lambda\cdot\mathcal{S} \iff \Theta_\lambda(e_i) < \Theta_\lambda(e_j) \tag{3.19}$$

The order relation computed for $\mathcal{A}'$ is given in Table 3.6.

Lemma 1

Correlation of Θ_λ and `prec()`

Show:

$$e_i \in \texttt{prec}(e_j) \implies \Theta_\lambda(e_i) < \Theta_\lambda(e_j)$$

Proof:

$\therefore$ *This follows directly from* (3.18) *since the Lamport Clock of an event is incremented by and only by members of that event's preceding event set.*

□

Theorem 6

Inconsistency of Lamport Clock and Cronos

Show:

$$\exists e_i, e_j \in \mathcal{S} \ni \langle e_i, e_j \rangle \in \lambda \cdot \mathcal{S} \wedge \langle e_j, e_i \rangle \in \chi \cdot \mathcal{S}$$

Proof: *by example*

1. $\exists e_2, e_8 \in \mathcal{S}$

   Figure 3.2.

2. $\langle e_8, e_2 \rangle \in \lambda \cdot \mathcal{A}' \wedge \langle e_2, e_8 \rangle \in \chi \cdot \mathcal{S}$

   Table 3.6, Table 3.2.

3. $\lambda \nparallel \chi$

   Step 1, Step 2, Property 4, Modus Ponens.

□

Corollary 2

Incorrectness of Lamport Clock vis-à-vis Cronos

Show:

$$\lambda \not\subseteq \chi$$

Proof:

$\therefore$ *This follows directly from Theorem 6, the contrapositive of Theorem 3 and Modus Ponens.*

□

Theorem 7

Correctness of Telos vis-à-vis Lamport Clock

Show:

$$\langle e_i, e_j \rangle \in \tau{\cdot}\mathcal{S} \implies \langle e_i, e_j \rangle \in \lambda{\cdot}\mathcal{S}$$

Proof:

1. $\langle e_i, e_j \rangle \in \tau{\cdot}\mathcal{S}$

   Hypothesis.

2. $e_i \in \Theta_\tau(e_j)$

   (3.14), Step 1, Modus Ponens.

3. $e_i \in \texttt{prec}(e_j)$

   (3.13), Step 2, Equality.

4. $\Theta_\lambda(e_i) < \Theta_\lambda(e_j)$

   Lemma 1, Step 3, Modus Ponens.

5. $\langle e_i, e_j \rangle \in \lambda{\cdot}\mathcal{S}$

   (3.19), Step 4, Modus Ponens.

6. $\langle e_i, e_j \rangle \in \tau{\cdot}\mathcal{S} \implies \langle e_i, e_j \rangle \in \lambda{\cdot}\mathcal{S}$

   Step 1, Step 5, Deduction.

□

Corollary 3

Consistency of Lamport Clock and Telos

Show:

$$\lambda \parallel \tau$$

Proof:

$\therefore$ *This follows directly from Theorem 7, Theorem 3, Theorem 2 and Modus Ponens.*

□

Theorem 8

Incorrectness of Lamport Clock vis-à-vis Telos

Show:

$$\exists e_i, e_j \in \mathcal{S} \ni \langle e_i, e_j \rangle \in \lambda \cdot \mathcal{S} \wedge \langle e_i, e_j \rangle \notin \tau \cdot \mathcal{S}$$

Proof: *by example*

1. $\exists e_0, e_5 \in \mathcal{A}'$

   Figure 3.2.

2. $\langle e_0, e_5 \rangle \in \lambda \cdot \mathcal{A}' \wedge \langle e_0, e_5 \rangle \notin \tau \cdot \mathcal{A}'$

   Table 3.6, Table 3.4.

3. $\lambda \not\subseteq \tau$

   Step 1, Step 2, Property 5, Modus Ponens.

□

Corollary 2 and Theorem 8 prove that the Lamport Clock protocol is not a correct temporal or causal order protocol, respectively. Intuitively, the temporal incorrectness is due to the different rates at which process clocks run, while the causal incorrectness is due to the fact that a lack of a computational connection between events does not cause their

Event	$\Theta_\nu(e)$
e_0	$(1,0,0)$
e_1	$(2,1,0)$
e_2	$(3,1,0)$
e_3	$(4,1,0)$
e_4	$(0,1,0)$
e_5	$(0,2,0)$
e_6	$(0,3,1)$
e_7	$(0,0,1)$
e_8	$(0,2,0)$
e_9	$(3,0,1)$

Table 3.7: VecClk Ordinals for $\mathcal{A}'$

$\nu \cdot \mathcal{A}'$			
$\langle e_0, e_1 \rangle$	$\langle e_0, e_2 \rangle$	$\langle e_0, e_3 \rangle$	$\langle e_0, e_9 \rangle$
$\langle e_1, e_2 \rangle$	$\langle e_1, e_3 \rangle$	$\langle e_1, e_9 \rangle$	
$\langle e_2, e_3 \rangle$	$\langle e_2, e_9 \rangle$		
$\langle e_4, e_1 \rangle$	$\langle e_4, e_2 \rangle$	$\langle e_4, e_3 \rangle$	$\langle e_4, e_9 \rangle$
$\langle e_5, e_6 \rangle$			
$\langle e_7, e_8 \rangle$	$\langle e_7, e_9 \rangle$	$\langle e_7, e_6 \rangle$	
$\langle e_8, e_9 \rangle$			

Table 3.8: VecClk Order Relation for $\mathcal{A}'$

ordinals to be incomparable. Both issues are addressed by the design of the next protocol. The causal order problem is also discussed in [30, 34].

Finally, as proven by Theorem 7, the Telos protocol is correct with regard to the Lamport Clock protocol, a result also obtained in [30]. This result also corresponds to the Clock Condition in [20], which, it should be noted, was the main goal of that paper, and not order consistency and correctness as discussed here.

3.3.2 Vector Clock (ν)

In the Vector Clock protocol, denoted ν and described in [10], each process maintains a vector clock, which is an array of Lamport Clocks. This array has an element for each process in the system, indexed by the process identifier. Intuitively, the j^{th} element in the vector clock for process P_i is the greatest Lamport Clock value for process P_j which P_i has encountered in previous events.

The event ordinal is the process vector clock at the time of the event. It is computed like that of the Lamport Clock protocol, except that each event updates an array. Letting Θ_ν^j

denote the j^{th} element of the event ordinal, and:

$$p := \texttt{pid}(e_i)$$
$$m^j := \texttt{max}\{\Theta_\nu^j(e_k), e_k \in \texttt{prec}(e_i)\}$$

then the event ordinal is computed:

$$\Theta_\nu^j(e_i) := \begin{cases} m^j & j \neq p \\ m^j + 1 & j = p \end{cases} \tag{3.20}$$

The event ordinals computed by the Vector Clock protocol for $\mathcal{A}'$ (Figure 3.2) are shown in Table 3.7.

Two event ordinals $\Theta_\nu(e_i)$ and $\Theta_\nu(e_j)$ are compared with the following set of equations:

$$\begin{aligned} &\Theta_\nu(e_i) \leq \Theta_\nu(e_j) \iff \forall k, \Theta_\nu^k(e_i) \leq \Theta_\nu^k(e_j) \\ &\Theta_\nu(e_i) < \Theta_\nu(e_j) \iff \Theta_\nu(e_i) \leq \Theta_\nu(e_j) \wedge \exists l \ni \Theta_\nu^l(e_i) < \Theta_\nu^l(e_j) \\ &\Theta_\nu(e_i) = \Theta_\nu(e_j) \iff \forall k, \Theta_\nu^k(e_i) = \Theta_\nu^k(e_j) \end{aligned} \tag{3.21}$$

The order relation for this protocol, $\nu{\cdot}\mathcal{S}$, is computed by comparing the ordinals for each pair of events:

$$\langle e_i, e_j \rangle \in \nu{\cdot}\mathcal{S} \iff \Theta_\nu(e_i) < \Theta_\nu(e_j) \tag{3.22}$$

The order relation computed by the Vector Clock protocol for $\mathcal{A}'$ is given in Table 3.8.

Lemma 2

Correlation of Θ_ν and $\texttt{prec}^*()$

Show:

$$\Theta_\nu(e_i) < \Theta_\nu(e_j) \iff e_i \in \texttt{prec}^*(e_j)$$

Proof:

$\therefore$ *This follows directly from* (3.20) *since the Vector Clock of an event is incremented by and only by members of that event's preceding event set.*

□

Theorem 9

Correctness of Vector Clock vis-à-vis Cronos

Show:

$$\langle e_i, e_j \rangle \in \nu \cdot \mathcal{S} \implies \langle e_i, e_j \rangle \in \chi \cdot \mathcal{S}$$

Proof:

1. $\langle e_i, e_j \rangle \in \nu \cdot \mathcal{S}$

   Assumption.

2. $\Theta_\nu(e_i) < \Theta_\nu(e_j)$

   (3.19), Step 1, Modus Ponens.

3. $e_i \in \texttt{prec}^\star(e_j)$

   Lemma 2, Step 2, Modus Ponens.

4. $\texttt{time}(e_i) < \texttt{time}(e_j)$

   Property 3, Step 3, Modus Ponens.

5. $\langle e_i, e_j \rangle \in \chi \cdot \mathcal{S}$

   (3.12), Step 4, Modus Ponens.

6. $\langle e_i, e_j \rangle \in \nu \cdot \mathcal{S} \implies \langle e_i, e_j \rangle \in \chi \cdot \mathcal{S}$

   Step 1, Step 5, Deduction.

□

Corollary 4

Consistency of Vector Clock and Cronos

Show:

$$\nu \parallel \chi$$

Proof:

$\therefore$ *This follows directly from Theorem 9, Theorem 3 and Modus Ponens.*

□

Theorem 10

Correctness of Vector Clock vis-à-vis Telos

Show:

$$\langle e_i, e_j \rangle \in \nu \cdot \mathcal{S} \implies \langle e_i, e_j \rangle \in \tau \cdot \mathcal{S}$$

Proof:

1. $\langle e_i, e_j \rangle \in \nu \cdot \mathcal{S}$

   Assumption.

2. $\Theta_\nu(e_i) < \Theta_\nu(e_j)$

   (3.19), Step 1, Modus Ponens.

3. $e_i \in \texttt{prec}^*(e_j)$

   Lemma 2, Step 3, Modus Ponens.

4. $\langle e_i, e_j \rangle \in \tau \cdot \mathcal{S}$

   (3.14), Step 3, Modus Ponens.

5. $\langle e_i, e_j \rangle \in \nu \cdot \mathcal{S} \implies \langle e_i, e_j \rangle \in \tau \cdot \mathcal{S}$

   Step 1, Step 4, Deduction.

□

Theorem 11

Correctness of Telos vis-à-vis Vector Clock

Show:

$$\langle e_i, e_j \rangle \in \tau \cdot \mathcal{S} \implies \langle e_i, e_j \rangle \in \nu \cdot \mathcal{S}$$

Proof:

1. $\langle e_i, e_j \rangle \in \tau \cdot \mathcal{S}$

   Assumption.

2. $e_i \in \Theta_\tau(e_j)$

   (3.14), Step 1, Modus Ponens.

3. $e_i \in \mathtt{prec}^\star(e_j)$

   (3.13), Step 2, Modus Ponens.

4. $\Theta_\nu(e_i) < \Theta_\nu(e_j)$

   Lemma 2, Step 3, Modus Ponens.

5. $\langle e_i, e_j \rangle \in \nu \cdot \mathcal{S}$

   (3.22), Step 4, Modus Ponens.

6. $\langle e_i, e_j \rangle \in \chi \cdot \mathcal{S} \implies \langle e_i, e_j \rangle \in \nu \cdot \mathcal{S}$

   Step 1, Step 5, Deduction.

□

Corollary 5

Equivalence of Vector Clock and Telos

Show:

$$vclk \equiv telos$$

Proof:

$\therefore$ *This follows directly from Theorem 10 and Theorem 11.*

□

Corollary 6

Consistency of Vector Clock and Telos

Show:

$$\nu \parallel \tau$$

Proof:

$\therefore$ *This follows directly from Theorem 10 (or Theorem 11), Theorem 3 and Modus Ponens.*

□

Theorem 9 proves that Vector Clock is a correct temporal order protocol and Corollary 5 proves that it is equivalent to Telos, the reference causal order protocol. The latter result was also demonstrated in [30].

Vector Clock has two main drawbacks. In [8], it was shown that the size of the array needed to compute a valid causal order was $O(n)$, where *n* is the number of processes involved in the computation. This means that the size of the clock array can be a problem if the number of processes involved in the computation is large.

The other drawback is the need to rely on data from other processes in computing the event ordinal. If any of the other processes is faulty or malicious, it may provide invalid data which would, in effect, slow down or speed up the vector clock for subsequent events.

3.3.3 Efficient Order Protocols

Several protocols have been designed to reduce the amount of overhead in the Vector Clock protocol while still maintaining its correctness. In the protocol presented in [35], each process records the last vector clock sent to each other process in the system. When a message is sent, the sender computes the difference between its current vector clock and the last one sent to the receiving process. This data, a set of $\langle pid, clock \rangle$ tuples, is included in the message as its ordinal; the 'last-sent' vector clock for this process is also updated.

When a process receives a message, it updates its vector clock according to the message ordinal. For example, the ordinal for message m_7 in $\mathcal{A}'$ would be $\{\langle 2,1\rangle\}$, while the ordinal for message m_2 would be $\{\langle 0,3\rangle,\langle 1,1\rangle\}$.

This protocol increases the amount of data transmitted for each updated Lamport Clock, but as long as relatively few of the clock elements have changed since the last message sent to the receiver, this approach will reduce the size of the message. The disadvantage of this approach is that it requires more local storage since each process must store the last vector clock sent to all other processes; it also requires more computation to determine the message ordinal.

In the protocol described in [3, 11], the message ordinal contains only the sender's element in the process vector clock - effectively its own Lamport Clock. The other members are stored by the process, indexed by the message ordinal; this data can be accessed after or even during the protocol execution to compute the causal order of events. For example, the ordinal for message m_2 in $\mathcal{A}'$ would be 3, and the ordinal for message m_4 would be 1.

This protocol is as efficient as the Lamport Clock protocol in terms of message overhead, but also requires local storage. The storage requirement can be even greater for a large computation because the stored array of vector clocks would be equal to the number of messages transmitted during the protocol execution rather than to the number of other processes in the computation. The other disadvantage is the causal order relation cannot be locally computed - if one process fails, significant parts of the order relation may be lost.

The main advantage of this protocol is that the order relation is computed only if it is needed. One of the cited applications in [11] is a debugger for distributed computations that runs only in the event of a system crash.

3.3.4 Order-Inducing Protocols

Order-inducing protocols determine event order actively, in contrast to order-computing protocols which detect event order passively. Order-inducing protocols contain a synchro-

nizing mechanism, such as a process group leader, which effectively controls the order of events in a system execution. To control order, processes normally queue messages upon reception for later processing after the order has been determined by the synchronizing mechanism.

Because of the synchronization overhead, order-inducing protocols are less efficient, but they may be required for those computations in which event order must be at least partially deterministic. Examples of this include replicated database or filesystem operations.

ABcast (μ)

The ABcast (Atomic Broadcast) protocol, part of the ISIS suite of protocols [6] and denoted μ, is a group broadcast protocol that supports atomic, ordered delivery of messages among the processes in the group. The order property, of interest here, guarantees that each process in the group sees all intra-group messages in the same order. ABcast has neither internal events nor separate send and receive events, and so messages and events are synonymous ($m_i \leftrightarrow e_i$).

A message sender broadcasts its message to all processes in the group. When a message is received, the receiver places it in a holding queue for that group and temporarily assigns it the highest number (lowest priority) of all messages in that queue. It sends the concatenation of that priority and its process identifier to the sender. (The process identifier acts as a tie-breaker.)

When the sender receives priorities for the message from all group members, it broadcasts the highest number to the other processes; each process then assigns this priority to the message and queues it for processing according to the new priority.

Letting $\texttt{prio}_k(m_i)$ represent the priority for message m_i assigned by process P_k, the ordinal for this protocol is:

$$\Theta_\mu(m_i) = \texttt{max}\{\texttt{prio}_k(m_i) \parallel k, \forall k\} \tag{3.23}$$

The order relation is computed as follows:

$$\langle m_i, m_j \rangle \in \mu\cdot\mathcal{S} \iff \Theta_\mu(m_i) > \Theta_\mu(m_j) \tag{3.24}$$

ABcast is a synchronous protocol in that the order in which messages are processed at each process is unrelated to the order in which they were received by that process. The sender of each message is the synchronizing agent. Note that the priority associated with each message essentially functions as a timestamp with numerically greater priorities corresponding to later times.

PSync (ψ)

The PSync (Pseudo-Synchronous) protocol [26], denoted ψ, is a group broadcast protocol similar to ABcast, but message processing is based on causal order rather than an absolute priority or timestamp. As with ABcast, events and messages are synonymous.

Each process maintains a directed acyclic graph of the protocol execution; the graph vertices are the messages which the process has sent or received during the execution, and edges are drawn to a vertex from its immediate predecessor(s). A vertex with no 'incoming' edge is the root vertex, and vertices with no 'outgoing' edges are the leaf vertices. The function $\texttt{pleaf}(P_i)$ returns the current leaf vertex set for process P_i. A leaf vertex set is

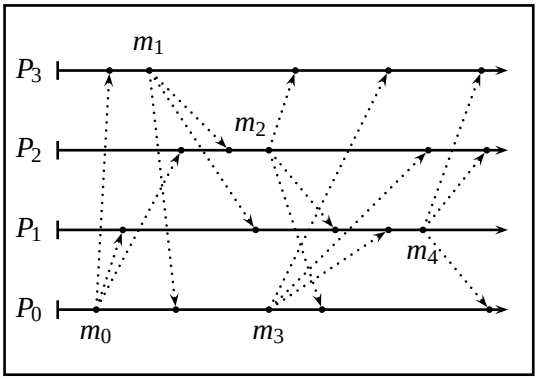

Figure 3.3: System $\mathcal{B}$

message	$\texttt{mleaf}(m)$
m_0	$\{\emptyset\}$
m_1	$\{m_0\}$
m_2	$\{m_1\}$
m_3	$\{m_1\}$
m_4	$\{m_2, m_3\}$

Figure 3.4: Msg LV Sets for $\mathcal{B}$

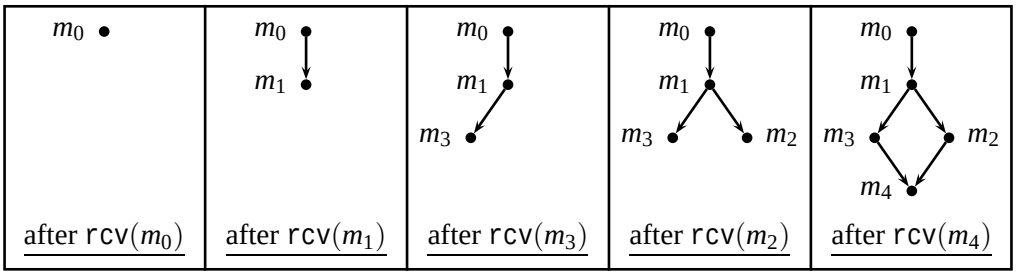

Figure 3.5: P_3 Directed Acyclic Graph for $\mathcal{B}$

also associated with each message; the function $\texttt{mleaf}(m_i)$ returns the leaf vertex set for message m_i.

The process leaf vertex set is initially empty, and is recomputed after each message send and receive:

$$\texttt{pleaf}(P_i) := \begin{cases} \{m_i\}; & \texttt{snd}(m_i) \\ \texttt{pleaf}(P_i) \cup \{m_i\} - \texttt{mleaf}(m_i); & \texttt{rcv}(m_i) \end{cases} \tag{3.25}$$

That is, when a process sends message m_i, it resets its leaf vertex set to m_i. When a process receives message m_i, the new message is queued until the process has received all of the messages in $\texttt{leaf}(m_i)$; the process's leaf vertex set is then modified by adding m_i and subtracting the members of $\texttt{mleaf}(m_i)$.

To better understand PSync process graphs, consider $\mathcal{B}$ in Figure 3.3, which depicts four processes and five broadcast messages. The graphics for this example are similar to those in the preceding examples but the only events are message transmissions (labeled m_i). The leaf vertex sets for each message are shown in Figure 3.4.

Figure 3.5 depicts the process directed graph for process P_3 just after receiving each message. Note that each process will compute the same directed graph, but may "draw" it in a different order. For example, process P_0 would draw the edge from m_1 to m_3 before the edge from m_1 to m_2 because it sees m_3 first.

Letting $\texttt{mleaf*}(m_i)$ denote the transitive closure of $\texttt{mleaf}(m_i)$, the event ordinal

for PSync is computed:

$$\Theta_{\psi}(m_i) = \texttt{mleaf*}(m_i) \tag{3.26}$$

And the order relation is computed:

$$\langle m_i, m_j \rangle \in \psi{\cdot}\mathcal{S} \iff m_i \in \Theta_{\psi}(m_j) \tag{3.27}$$

While PSync is an order-inducing protocol like ABcast, it only orders causally related messages; other messages are processed asynchronously.

3.4 clio (κ)

clio is a protocol which supports the causal ordering of events in Forum. Forum consists of a set of protocols and server processes used by client processes to conduct transactions. The design of the main elements of Forum, including clio, is presented in chapter 4. This section contains a description of the order computation done by clio and an analysis of its order properties with regard to the reference order protocols.

3.4.1 clio Subtransactions

A transaction is the unit of electronic commerce in Forum. A transaction contains one or more subtransactions. Subtransaction y in transaction x has the unique identifier $x.y$. Each subtransaction is between two parties, and has the following phases: initialization, during which the two parties accept or reject the subtransaction; active during which the two parties exchange messages; and termination during which the two parties commit or abort the subtransaction. If both parties commit, they must agree on the causal order of the messages exchanged during the active phase.

The active phase of a subtransaction consists of a sequence of `t_data` messages, each of which has a unique identifier. Each process computes two arrays which contain the identifiers for each message in the subtransaction. In one array (`s_order`), the message

identifiers are ordered according to the sequence of message as seen by the process, while in the other array (`r_order`), the message identifiers are ordered from the viewpoint of the other process.

The `s_order` array is easy to compute; as a message is sent or received, its identifier is added to the array. The `r_order` array is computed using the `ack` field in the `t_data` message. This field is an acknowledgment of the most recently received but unacknowledged message in this subtransaction, and is set to that message's identifier.

When a `t_data` message is sent, its identifier is marked as unacknowledged. When a `t_data` message is received, its `ack` field is compared with the identifiers of all unacknowledged messages. If a match is found, then the identifiers of that message and any earlier unacknowledged messages are added to the `r_order` array in the order in which they were sent, followed by the identifier of the received message.

For example, figure 3.6 depicts two processes engaged in a subtransaction $p.q$ which contains five messages, which have identifiers '**u**', '**n**', '**i**', '**t**' and '**e**'. The order arrays for $p.q$ for process P_0 are shown in figure 3.7. Note that these two arrays are different for indices 2 and 3, and so P_0 sees the subtransaction message order as "unite" and P_1 sees the subtransaction message order as "untie". As long as the difference between "unite" and "untie" is inconsequential in the context of the particular subtransaction, the two processes can proceed to the third phase.

During the third phase of a subtransaction, each process reports its order arrays to the transaction manager, which cross-compares them – the `s_order` array of one process is compared to the `r_order` array of the other process, and vice versa. If the cross-comparisons are equal, then the subtransaction is marked as successful.

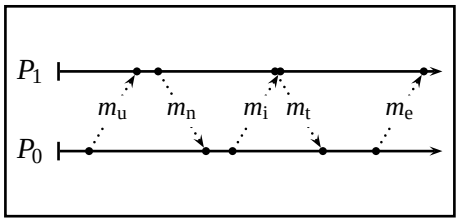

Figure 3.6: Subtransaction $p.q$

$s\_order_{p,q}$	u	n	i	t	e
$r\_order_{p,q}$	u	n	t	i	e

Figure 3.7: P_0 order arrays for $p.q$

3.4.2 clio Ordinal and Order Relation

Function 6

stid()

stid is a function from the set of messages to ordered pairs of natural numbers:

$$\texttt{stid} : \mathcal{S} \mapsto \mathbb{N} \times \mathbb{N}$$

$\texttt{stid}(m_i)$ returns the subtransaction identifier of m_i in the form $x.y$.

Function 7

s_ndx()

s_ndx is a function from the set of events to the set of natural numbers:

$$\texttt{s\_ndx} : \mathcal{S} \mapsto \mathbb{N}$$

constrained by:

$$\texttt{stid}(m_i) = \texttt{stid}(m_j) \wedge \texttt{s\_ndx}(m_i) = \texttt{s\_ndx}(m_j) \implies i = j \tag{3.28}$$

Note: s_ndx is also constrained by (3.30).

$\texttt{s\_ndx}(m_i)$ returns the index of m_i in the s_order array for the subtransaction $\texttt{stid}(m_i)$ as reported by the subtransaction creator.

Event	$\Theta_\kappa(m_a)$
m_u	$\langle p.q, 0, 0\rangle$
m_n	$\langle p.q, 1, 1\rangle$
m_i	$\langle p.q, 2, 3\rangle$
m_t	$\langle p.q, 3, 2\rangle$
m_e	$\langle p.q, 4, 4\rangle$

Table 3.9: clio Ordinals for $p.q$

$\kappa{\cdot}p.q$	
$\langle m_u, m_n\rangle$	$\langle m_u, m_i\rangle$
$\langle m_u, m_t\rangle$	$\langle m_u, m_e\rangle$
$\langle m_n, m_i\rangle$	$\langle m_n, m_e\rangle$
$\langle m_i, m_e\rangle$	
$\langle m_t, m_e\rangle$	

Table 3.10: clio Order Relation for $p.q$

Function 8

r_ndx()

r_ndx is a function from the set of events to the set of natural numbers:

$$\texttt{r\_ndx} : \mathcal{S} \mapsto \mathbb{N}$$

constrained by:

$$\texttt{stid}(m_i) = \texttt{stid}(m_j) \wedge \texttt{r\_ndx}(m_i) = \texttt{r\_ndx}(m_j) \implies i = j \tag{3.29}$$

$$\texttt{stid}(m_i) = \texttt{stid}(m_j) \wedge \texttt{s\_ndx}(m_i) < \texttt{s\_ndx}(m_j) \wedge \texttt{r\_ndx}(m_i) < \texttt{r\_ndx}(m_j)$$
$$\implies m_i \in \texttt{prec}^*(m_j) \tag{3.30}$$

$\texttt{r\_ndx}(m_i)$ returns the index of m_i in the r_order array for the subtransaction $\texttt{stid}(m_i)$ as reported by the subtransaction creator.

The event ordinal for a message m_i consists of the tuple containing the message's subtransaction identifier and its indices in the two order arrays.

$$\Theta_\kappa(m_i) = \langle \texttt{stid}(m_i), \texttt{s\_ndx}(m_i), \texttt{r\_ndx}(m_i)\rangle \tag{3.31}$$

The event ordinals computed by clio for subtransaction $p.q$ (figure 3.6) are shown in Table 3.9.

The message ordinal consists of the t_data message ack field which, as noted earlier, contains the unique identifier of the last message received from the other process.

Two clio ordinals $\Theta_\kappa(m_i)$ and $\Theta_\kappa(m_j)$ are compared as follows:

$$\begin{gathered}\Theta_\kappa(m_i) < \Theta_\kappa(m_j) \iff \\ \texttt{stid}(m_i) = \texttt{stid}(m_j) \wedge s\_ndx(m_i) < s\_ndx(m_j) \wedge r\_ndx(m_i) < r\_ndx(m_j)\end{gathered} \tag{3.32}$$

The order relation for clio is computed by comparing the ordinals for each pair of events:

$$\langle m_i, m_j \rangle \in \kappa{\cdot}\mathcal{S} \iff \Theta_\kappa(m_i) < \Theta_\kappa(m_j) \tag{3.33}$$

3.4.3 clio Consistency and Correctness

Lemma 3

Correlation of Θ_κ and $\texttt{prec*}()$

Show:

$$\langle m_i, m_j \rangle \in \kappa{\cdot}\mathcal{S} \implies m_i \in \mathit{prec}*(m_j)$$

Proof:

1. $\langle m_i, m_j \rangle \in \kappa{\cdot}\mathcal{S}$

   Assumption.

2. $\Theta_\kappa(m_i) < \Theta_\kappa(m_j)$

   Step 1, (3.33), Modus Ponens.

3. $\mathit{stid}(m_i) = \mathit{stid}(m_j) \wedge s\_ndx(m_i) < s\_ndx(m_j) \wedge r\_ndx(m_i) < r\_ndx(m_j)$

   Step 2, (3.32), Modus Ponens.

4. $m_i \in \mathit{prec}*(m_j)$

   Step 3, (3.30), Modus Ponens.

5. $\langle m_i, m_j \rangle \in \kappa{\cdot}\mathcal{S} \implies m_i \in \mathit{prec}*(m_j)$

   Step 1, Step 4, Deduction.

□

Theorem 12

Correctness of clio vis-à-vis Cronos

Show:

$$\langle m_i, m_j \rangle \in \kappa \cdot \mathcal{S} \implies \langle m_i, m_j \rangle \in \chi \cdot \mathcal{S}$$

Proof:

1. $\langle m_i, m_j \rangle \in \kappa \cdot \mathcal{S}$

   Assumption.

2. $m_i \in \texttt{prec}^*(m_j)$

   Lemma 3, Step 1, Modus Ponens.

3. $\texttt{time}(m_i) < \texttt{time}(m_j)$

   Property 3, Step 2, Modus Ponens.

4. $\langle m_i, m_j \rangle \in \chi \cdot \mathcal{S}$

   (3.12), Step 3, Modus Ponens.

5. $\langle m_i, m_j \rangle \in \kappa \cdot \mathcal{S} \implies \langle m_i, m_j \rangle \in \chi \cdot \mathcal{S}$

   Step 1, Step 4, Deduction.

□

Corollary 7

Consistency of clio and Cronos

Show:

$$\kappa \parallel \chi$$

Proof:

$\therefore$ *This follows directly from Theorem 12, Theorem 3 and Modus Ponens.*

□

Theorem 13

Correctness of clio vis-à-vis Telos

Show:

$$\langle m_i, m_j \rangle \in \kappa \cdot \mathcal{S} \implies \langle m_i, m_j \rangle \in \tau \cdot \mathcal{S}$$

Proof:

1. $\langle m_i, m_j \rangle \in \kappa \cdot \mathcal{S}$

   Assumption.

2. $m_i \in \texttt{prec*}(m_j)$

   Lemma 3, Step 1, Modus Ponens.

3. $\langle m_i, m_j \rangle \in \tau \cdot \mathcal{S}$

   Step 2, (3.14), Modus Ponens.

4. $\langle m_i, m_j \rangle \in \kappa \cdot \mathcal{S} \implies \langle m_i, m_j \rangle \in \tau \cdot \mathcal{S}$

   Step 1, Step 3, Deduction.

□

Corollary 8

Consistency of clio and Telos

Show:

$$\kappa \parallel \tau$$

Proof:

$\therefore$ *This follows directly from Theorem 13, Theorem 3 and Modus Ponens.*

□

Conclusions

By Theorem 12 and Corollary 7, clio is temporally correct and consistent, and by Theorem 13 and Corollary 8, clio is causally correct and consistent.

3.5 Summary

This chapter defines a formal model for determining the correctness and consistency of order protocols. This model consists of a system definition, functions and properties pertaining to system event order, and two reference protocols that are used to establish order correctness and consistency. Both temporal and causal order are addressed by the model.

The formal model was used to analyze several existing protocols, including Lamport Clock and Vector Clock. The efficiency of Vector Clock protocols was also discussed, and two order-inducing protocols, ABcast and Psync, were described using parts of the model to illustrate their ordering mechanisms.

Lastly clio, an order protocol for electronic commerce, was described, and the model was used to prove its temporal and causal order correctness and consistency.

Chapter 4

Design

4.1 Overview

design *n.* [24]

2. The invention and disposition of the forms, parts or details of something according to a plan.

This chapter describes the design of Forum, a system intended for general forms of electronic commerce. A design may be evaluated on the basis of function, properties and aesthetics. Here the emphasis will be on the first two criteria.

Electronic commerce systems have the same function as non-electronic ones: to exchange goods, services and financial instruments. To effect these exchanges in electronic commerce, computer processes, acting as agents for individuals and public or private organizations, send and receive messages using communication protocols in which the messages represent goods, services and financial instruments.

Systems for electronic commerce also normally provide a means to group related exchanges into a transaction, but this requires little more than a mechanism to assign unique transaction identifiers and the inclusion of the identifier in each related message.

Functionally, then, electronic commerce needs reliable delivery of messages, a naming service so that agents can obtain the network addresses of each other, and a service to assign unique transaction identifiers. The first and second are provided by basic TCP/IP, augmented by an X.500 or LDAP directory service, and the third is simple to provide.

Systems for electronic commerce support various properties for either individual messages or for the messages grouped into transactions. These properties are instantiated by the execution of a protocol designed for that purpose. Three types of properties are considered here:

Contractual properties are required because every commercial transaction is also a legal contract which imposes obligations on each participant. The contractual properties supported by Forum are:

agreement

All transaction participants must consent to the terms, conditions and obligations of the exchange.

consideration

All obligated transaction participants must receive something in the exchange.

obligation

All transactions include obligations which take effect simultaneously.

durability

The messages in a transaction may be recorded for quality control purposes.

Security properties are required because of the monetary value of commerce coupled with the security risks of computer networks. The security properties supported by Forum are:

privacy

The data in transaction messages may be viewed only by the receiver.

integrity

The receiver of a transaction message can verify the authenticity of the message data.

origin

The receiver of a transaction message can verify the identity of the sender; if the message corresponds to an obligation, the origin is non-repudiable.

uniqueness

The receiver of a transaction message can verify its newness.

Structural properties define the relationships among elements of a design, and so are applicable to most objects. The structural properties supported by Forum are:

quantitative scalability

The number of participants in a transaction is arbitrary.

qualitative scalability

The role of each participant in a transaction is arbitrary.

Further discussion and analysis of the supported properties may be found in chapter 2.

Existing electronic commerce frameworks focus primarily on security properties, with little support for contractual properties; structurally, they support two-party transactions, with a single arbiter to coordinate the transaction. Thus these frameworks are suited only to basic forms of commerce. By supporting the above properties, Forum can provide a uniform framework which will enable more general forms of electronic commerce than existing frameworks.

Forum is defined and analyzed in the following sections, beginning with the Forum system model, which provides an initial design point. Next are sections on the major Forum components, emphasizing the algorithms used to support the properties defined above.

One other property will be useful in the design of Forum: simplicity. When choices must be made among algorithms, protocols or system interfaces, Occam's razor will be the preferred design tool.

Perfection is reached, not when there is no longer anything to add, but when there is no longer anything to take away.

Antoine de Saint-Exupery

4.2 Forum System Model

The system model for Forum is based on a small number of design premises:

- Transactions are the *raison d'tre* of Forum.
- There are protocols to conduct transactions, communicate securely and obtain cryptographic keys.

- There are servers for transaction management, data storage and key storage.

4.2.1 Forum Transactions

Transactions are the units of commerce in Forum. The semantics of transactions are unspecified; they may concern negotiations to establish a business relationship or the exchange of goods or services or monetary instruments, or they may have only private significance.

Transactions are initiated and carried on by Forum client processes and are managed by a Forum server process. Each transaction has a unique identifier assigned by the managing server. Each message within a transaction has a unique identifier.

4.2.2 Forum Processes

Forum has client and server processes, each of which has a unique identifier and an identity certificate. Aside from these, Forum client processes are indistinctive. There are three types of servers:

Transaction servers manage transaction initiation and termination, and report transaction status. They also can be used to verify message validity and ordering. The former judgment can be used to authorize the exchange of items agreed to in the course of a transaction, while the latter can be used to establish a causal relationship among the parts of a transaction.

Durability servers provide a permanent record of the data for a transaction. Archival requirements are specific to each process and transaction, and so client processes are not required to use a durability server for all transactions.

Key servers issue and maintain symmetric cryptographic keys used for data privacy, integrity and origin, and asymmetric public-private keys used for data integrity and non-repudiation.

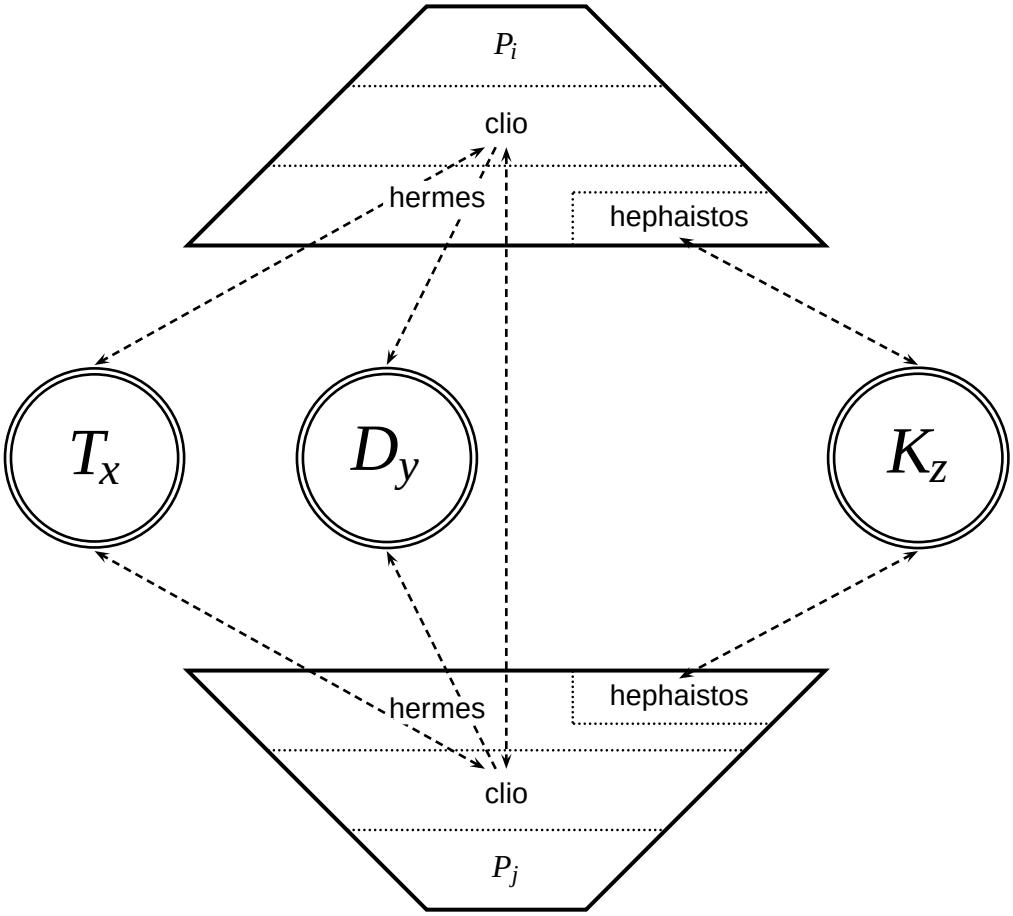

Figure 4.1: Forum Design Model

4.2.3 Forum Protocols

All processes communicate via reliable channels using one or more of the Forum protocols. The clio protocol supports the contractual and structural properties, while the hermes and hephaistos protocols support the security properties. hermes is used for secure communication among processes, while hephaistos is used to obtain cryptographic keys.

The relationship of these protocols is depicted in Figure 4.1. In this figure, P_i and P_j represent Forum client processes and the circles represent Forum server processes: T_x, a

transaction server, D_y, a durability server, and K_z, a key server.

In the client processes, the second layer contains the contractual protocol clio and the bottom layer contains the security protocols hermes and hephaistos. Note that the hephaistos protocol is not invoked directly by clio. It is a subordinate protocol invoked by hermes to obtain keys.

The arrows denote communication paths among the processes using the designated protocol. To enhance clarity, some less important paths have been omitted, and the security layer is not shown for the server processes.

4.3 clio

clio, the daughter of Zeus and Mnemosyne, was the Muse of History. It is the main protocol in Forum and is used by processes to conduct transactions. It supports the contractual properties of agreement, consideration, obligation and durability, and the structural properties of quantitative and qualitative scalability.

Section 4.3.1 analyzes the algorithms that can be used to support these properties, section 4.3.2 provides an overview of clio, and sections 4.3.3 and 4.3.4 describe clio in terms of valid message types and sequences.

4.3.1 Algorithms

The discussion of commercial transaction properties in chapter 2 analyzes the correspondence between these properties and the ACID properties of database transactions. Algorithms which support the database transaction properties will be used as the primary basis for the commercial transaction properties.

Agreement

The database property of atomicity does not correspond directly to agreement, but atomicity depends upon agreement (in this case, of the database resource managers), and so

contractual agreement can be supported using a database atomicity algorithm. Atomicity is achieved with a commit protocol which allows the resource managers to signal their state for a transaction when it ends. If all signal affirmatively, the transaction is deemed successful.

The most common type of commit protocol is two-phase commit[4]. When the transaction ends, the transaction manager sends prepare requests to each resource manager involved in the transaction. The resource manager sends back a commit or abort message, depending on whether or not it deemed that the transaction execution was correct. Correctness is domain specific; for a scheduler it might mean that the transaction execution was serializable, or for a storage manager, it might mean that the transaction data was stored on the disk drive.

If all resource managers vote to commit, the transaction manager marks the transaction as committed and sends commit messages back to each resource manager to notify them of the successful termination. If not all vote to commit, the transaction is marked as aborted and abort messages are sent instead, so that the resource managers can clean up their state and free the resources used for the transaction.

For commercial transactions, agreement is required of all the participants in order for the transaction to be successful. Although the database resource managers are only checking for failures rather than expressing volition, the end result is the same: that all parties to the transaction signal that it is valid. Thus, agreement is supported in Forum with a similar commit protocol among the transaction participants.

Consideration

Consideration has no analogue in database transactions, but consideration is also known as *quid pro quo* (that for which). The *pro* is causal, and so a causal order protocol should be sufficient to support this property.

In chapter 3, causal and temporal order protocols are described and analyzed, and a

protocol is presented which allows two processes to compute the partial order of messages between them. clio uses this protocol to support consideration.

Obligation

The database property isolation is similar to obligation in that transaction state changes for both are to be effective simultaneously. Isolation in a database system is supported using concurrency control [19, 4]. The central aim of concurrency control is to achieve a schedule of operations such that each transaction appears as though it were the only transaction in the system during its execution. This is termed serializability.

To achieve serializability, conflicting operations must be scheduled, usually with a locking algorithm; this forces transactions to acquire a lock before accessing a data item. The lock will not be granted unless the mode of the access will not conflict with an existing lock on the item. A popular lock algorithm is two-phase locking, so called because transactions must acquire all locks before releasing any of them.

In commercial transactions, obligation is mutual: all transaction obligations are binding or none are. An obligation in an electronic commerce transaction is equivalent to a write operation in a database transaction, in that it changes state. Locking or any other concurrency control algorithm is more difficult in electronic commerce because the transaction server in electronic commerce cannot have the same degree and kind of control over access to its domain as does the transaction manager in a database system.

Fortunately, obligation can be supported by a simpler locking discipline. Obligations in electronic commerce are incurred by and only by messages in a successful transaction, so that an obligation should be binding only after the message containing the obligation has been validated by the transaction server.

Thus, obligation can be supported if the transaction server changes transaction state atomically and confirms transaction message validity only for successful transactions. These restrictions allow the transaction state to serve as a lock. When the state of a trans-

action becomes committed, the lock is released and all the obligations in that transaction simultaneously become binding.

Note that this discipline requires that the validity of a transaction message be confirmed with the transaction server before any obligation implied by that message is honored. This procedure is neither onerous nor unusual, since it is similar to that employed by businesses before honoring checks or credit card charges.

Durability

Durability has the same meaning for both database and commercial transactions: the data changes of a successful transaction will persist in spite of system, device or media failures. In databases, this type of memory is known as stable storage.

There are many kinds of stable storage. Media and device failures are usually corrected with data replication, either at different locations on the same device or on multiple devices. System failures can be masked by recording changes in a log or by making changes to a copy of the database; in the event of a system failure, the state in the log or the database copy is used when the system is restarted to update the database.

Durability in database systems is obtained by implementing a storage manager with stable storage, and then including the storage manager in the commit protocol. Durability in Forum is achieved in a similar fashion.

Quantitative Scalability

In database systems, there are several ways to decompose transactions to increase their 'arity' [14]. The main ones are:

chained transactions separate transactions into sequences of subtransactions. As each subtransaction commits, data items that are no longer needed are unlocked while the remaining data items are passed along to the next subtransaction in the chain.

nested transactions permit one or more transactions to be included in a parent transaction. Each child transaction can in turn have children, so that the transaction structure is a tree. As with chained transactions, nested transactions are intended to provide structured save points, but nested subtransactions do not share data items.

distributed transactions allow data to be stored across multiple systems. A distributed transaction is divided according to where its data items are stored. Logically, however, this is still a single transaction.

None of these appear to be analogous to commercial transactions. The chained model is the simplest but multi-party commercial transactions may not be sequential nor would they bequeath data items sequentially. Commercial transactions also would not need the partial commitment of the chained and nested transaction models. And the distributed model enables multiple database subsystems rather than multiple transaction participants.

Instead, to support quantitative scalability the transaction model in Forum uses a simple decomposition of a transaction into a variable number of discrete, indivisible and equipollent[1] two-party subtransactions.

Qualitative Scalability

Database transaction are also not a good model for qualitative scalability. Database users do not directly interact, and the processing components of a database system are fixed (query manager, storage manager, lock manager, ...), even in the distributed case.

In commercial transactions, qualitative scalability is the capability to allow variable roles for transaction participants. A participant's role in a commercial transaction is defined by what she gives, receives, controls or records. In electronic commerce, transfers, approval and knowledge are determined by the sequence and contents of messages sent or received by each participant. Thus, qualitative scalability can be supported by simply not constraining transaction message sequences or contents.

[1] Equal in power, effectiveness or significance.

4.3.2 Protocol Overview

clio is used by Forum client processes to communicate with a transaction server to initiate and terminate transactions and subtransactions, and to communicate with other Forum client processes during subtransactions. clio can also be used to communicate with a durability server to save the subtransaction data.

Each transaction consists of an arbitrary number of subtransactions. Each subtransaction is between two processes. The contents and order of messages in a subtransaction are unrestricted. Subtransactions can be created in a new transaction by any process and in an existing transaction by any process already participating in that transaction. The terminal state for transactions or subtransactions is either `void`, `committed` or `aborted`, signifying unsuccessful initiation, success or failure, respectively.

For each subtransaction, clio maintains ordering information in each message that can be used to deduce causal relationships among the messages in a transaction. At the conclusion of a transaction, clio computes transaction success or failure. And, for a successful transaction, clio provides message validity status that can be used by processes to verify the obligations incurred in that transaction.

Initiation

The two processes interact with the transaction server to create a subtransaction in a new or existing transaction. To create a subtransaction, a client process sends a message to the transaction server specifying the identity of the other process. The transaction server then sends join requests to both processes, which they can accept or reject. If both accept, the subtransaction becomes active. Otherwise, the subtransaction is void.

Active

The two processes exchange data messages with each other using a full-duplex protocol. Each message includes both data and causal information. The causal information

is used by both processes during the subtransaction to compute the partial order of the messages sent and received by each process.

The causal data in each message consists of the message identifiers of the last message sent and received for the subtransaction by the message sender. Message identifiers are unique, and are computed as the cryptographic checksum of the message data.

Termination

The two processes interact with the transaction server to conclude the subtransaction. Either process can end a subtransaction by sending a commit or abort request to the transaction server. The transaction server then sends an end request to the other subtransaction process, which will respond with its own commit or abort request. The causal data is included in a commit request. If both processes commit and their causal data are in agreement, the subtransaction is successfully committed; otherwise it is aborted.

When a transaction contains no active subtransactions, the transaction server computes the final status of the transaction. The transaction status is `committed` if all subtransactions are either `committed` or `void`; otherwise it is `aborted`.

Postlude

After the subtransaction has been concluded, processes can query the transaction server to determine subtransaction and transaction status and, after the parent transaction has been concluded, message order and validity as well.

clio supports durability by copying subtransaction data to a durability server. Subtransaction data recording is optional, and either client process may choose to record the data. The client chooses the durability server to use for each subtransaction, and so can use the kind and degree of stable storage appropriate for the subtransaction data.

The durability server, by both design and implementation, is simply a clio client, but

a passive one which does not initiate or end subtransactions, and neither rejects nor aborts subtransactions unless there is a storage or authorization failure.

If durability is requested for a subtransaction, clio does the following additional processing:

Initiation

The clio client process begins an adjunct subtransaction, known as the echo subtransaction, naming a durability server as the other client.

Active

The clio client process copies all data sent and received during the subtransaction to the durability server as part of the echo subtransaction. The data sent to the durability server is encrypted so that the durability server cannot read the data.

Termination

The clio client process will end the echo subtransaction when the main subtransaction is ended. The client process will either commit or abort both subtransactions.

Postlude

The clio client process can retrieve subtransaction data from a durability server if the parent transaction was committed.

Note that the integration of the durability server into the transaction as "yet another clio client" guarantees that the transaction will be unsuccessful if the transaction data cannot be effectively recorded.

4.3.3 Protocol Message Types

clio defines the following messages, grouped according to subtransaction phase.

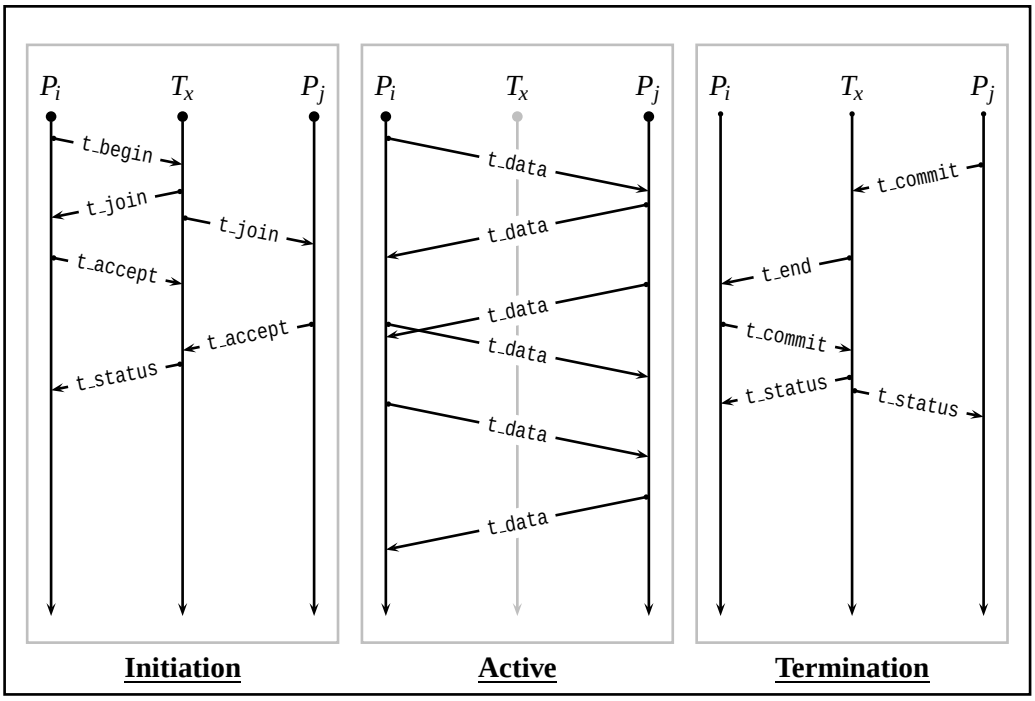

Figure 4.2: clio Transaction Phase Messages

Initiation Messages

`t_begin(`*`xid, uid, sbxdata`*`)`

is sent by a client process to the clio transaction server to begin a new subtransaction in either an existing or a new transaction. The message fields are:

xid the transaction identifier (0 to start a new transaction);

uid the identifier of the other process;

sbxdata subtransaction specific data (returned to the process in the `t_join` message).

Note: Any client may begin a new transaction while only a participant may create a subtransaction in an existing transaction.

`t_join(`*`xid, subxnum, uid, sbxdata`*`)`

is sent by the clio transaction server to both clients (the creator and the other process) in response to a `t_begin` message to invite them to participate in a new subtransaction. The message fields are:

xid the transaction identifier;

subxnum the subtransaction number;

uid the identifier of the other process;

sbxdata included from the `t_begin` message to be interpreted by the processes.

Note: The `t_join` message is also sent to the subtransaction creator because it contains the new subtransaction number.

`t_accept(`*`xid, subxnum`*`)`

is sent by a client to the clio transaction server in response to a `t_join` message to accept a new subtransaction. The message fields are:

xid the transaction identifier;

subxnum the subtransaction number.

Note: The subtransaction becomes active if both clients accept the join invitation.

`t_reject(`*`xid, subxnum`*`)`

is sent by a client to the `clio` transaction server in response to a `t_join` message to reject a new subtransaction. The message fields are:

xid the transaction identifier;

subxnum the subtransaction number.

Note: The subtransaction becomes void if either client rejects the join invitation.

Active Messages

`t_data(`*`xid, subxnum, data, ack, prec`*`)`

is sent by a client to the other process in an active subtransaction to exchange transaction data. The message fields are:

xid the transaction identifier;

subxnum the subtransaction number;

data the subtransaction data;

ack the identifier of the most recent message received by the sender in this subtransaction or 0, if no messages have been received yet;

prec the identifier of the most recent message transmitted by the sender in this subtransaction or 0, if no messages have been transmitted yet.

Termination Messages

`t_end(`*`xid, subxnum`*`)`

is sent by the clio transaction server to a client to request subtransaction termination. The message fields are:

xid the transaction identifier;

subxnum the subtransaction number.

`t_commit(`*`xid, subxnum, sndord, rcvord`*`)`

is sent by a client to the clio transaction server to request subtransaction commitment, either asynchronously or in response to a `t_end` message. The message fields are:

xid the transaction identifier;

subxnum the subtransaction number;

sndord an array of message identifiers defining the message order from the point of view of this client;

rcvord an array of message identifiers defining the computed message order from the point of view of the other subtransaction client.

Note: The subtransaction is committed if and only if and only if both clients commit and their orders agree. The orders are cross-compared - one client's `sndord` array is compared to the other client's `rcvord` array.

`t_abort(`*`xid, subxnum`*`)`

is sent by the client to the clio transaction server to request subtransaction abortion, either asynchronously or in response to a `t_end` message. The message fields are:

xid the transaction identifier;

subxnum the subtransaction number.

Note: The subtransaction is aborted if either client aborts the subtransaction or if their committed order arrays differ.

Postlude Messages

`t_qstatus(`*`xid, subxnum`*`)`

is sent by a client to the clio transaction server to request transaction or subtransaction status. The message fields are:

xid the transaction identifier;

subxnum the subtransaction number - if zero, the transaction status is returned.

`t_status(`*`xid, subxnum, state`*`)`

is sent by the clio transaction server to a client either asynchronously or in response a `t_qstatus` message to report subtransaction or transaction status. The message fields are:

xid the transaction identifier;

subxnum the subtransaction number - if zero, the transaction status is being returned.

state the status, which will be either `void`, `active`, `committed` or `aborted`.

Note: This message is sent asynchronously by the clio transaction server to a client to activate a subtransaction if both processes accept the join invitation. The message is sent only to the subtransaction creator. The other party is 'activated' by receiving a subtransaction data message from the creator.

`t_qorder(`*xid, subxnum, mid0, mid1*`)`

is sent by a client to the clio transaction server to request message ordering data for a subtransaction in a committed transaction. The message fields are:

xid the transaction identifier;

subxnum the subtransaction number;

mid0 the identifier of a message in the specified subtransaction;

mid1 the identifier of another message in the specified subtransaction.

`t_order(`*xid, subxnum, mid0, mid1, order*`)`

is sent by the clio transaction server to a client in response to a `t_qorder` message to report message ordering data. The message fields are:

xid the transaction identifier;

subxnum the subtransaction number;

mid0 the first identifier from the `t_qorder` message;

mid1 the second identifier from the `t_qorder` message;

order an integer which is less than zero if the first message causally precedes the second message, greater than zero if the second message causally precedes the first message, and zero if they are simultaneous.

Note: One message will be considered as causally preceding another message if the identifier of the first message occurs before that of the second message in both committed order arrays, or as causally succeeding the other message if the opposite is true; otherwise they are considered to be simultaneous.

`t_qvalid(`*`xid, subxnum, mid`*`)`

is sent by a client to the clio transaction server to request message validity data for a subtransaction in a committed transaction. The message fields are:

xid the transaction identifier;

subxnum the subtransaction number;

mid the identifier of the message whose validity is being verified.

`t_valid(`*`xid, subxnum, mid, valid`*`)`

is sent by the clio transaction server to a client in response to a `t_qvalid` message to report message validity data. The message fields are:

xid the transaction identifier;

subxnum the subtransaction number;

mid the identifier of the message whose validity is being reported;

valid a boolean indicating if the message is valid in the specified subtransaction.

Note: A message will be considered valid if its message identifier is found in the committed order arrays for the specified subtransaction.

`t_qecho(`*`xid, subxnum, mid`*`)`

is sent by a client process to a durability server to retrieve a previously stored message. The message fields are:

xid the transaction identifier;

subxnum the subtransaction number;

mid the identifier of the requested message.

`t_echo(`*`xid, subxnum, mid, xdata`*`)`

is sent by a durability server to a client process in response to a `t_qecho` message to return the requested data. The message fields are:

xid the transaction identifier;

subxnum the subtransaction number;

mid the identifier of the requested message;

data the requested subtransaction data (encrypted).

4.3.4 Protocol Message Sequences

Valid message sequences during the first three phases of a subtransaction are depicted in Figure 4.2. In this figure, P_i and P_j are basic processes, while T_x is the transaction server. Note the `t_status` message at the end of the initiation phase is sent to P_i, the subtransaction creator, who becomes active; the first message sent by P_i will cause P_j to become active.

A more detailed example with four clients and three subtransactions (from the same parent transaction) is depicted in Figure 4.3. This example shows two clients, P_i and P_j, forming a new subtransaction. Next P_j first fails to form a subtransaction with P_l and then succeeds with P_k. Lastly, P_j completes the initial subtransaction with P_i.

Note that a void subtransaction does not end the transaction. A transaction can be committed as long as all its subtransactions complete in either the `void` or `committed` states.

A final example, Figure 4.4, shows a subtransaction between processes P_i and P_j which is being echoed to a durability server, process D_y. The messages that are part of the echo subtransaction are labeled accordingly. Note that the data transfer is one-way (from P_j to D_y only), and the P_j delays accepting the first `t_data` message from P_i until successfully

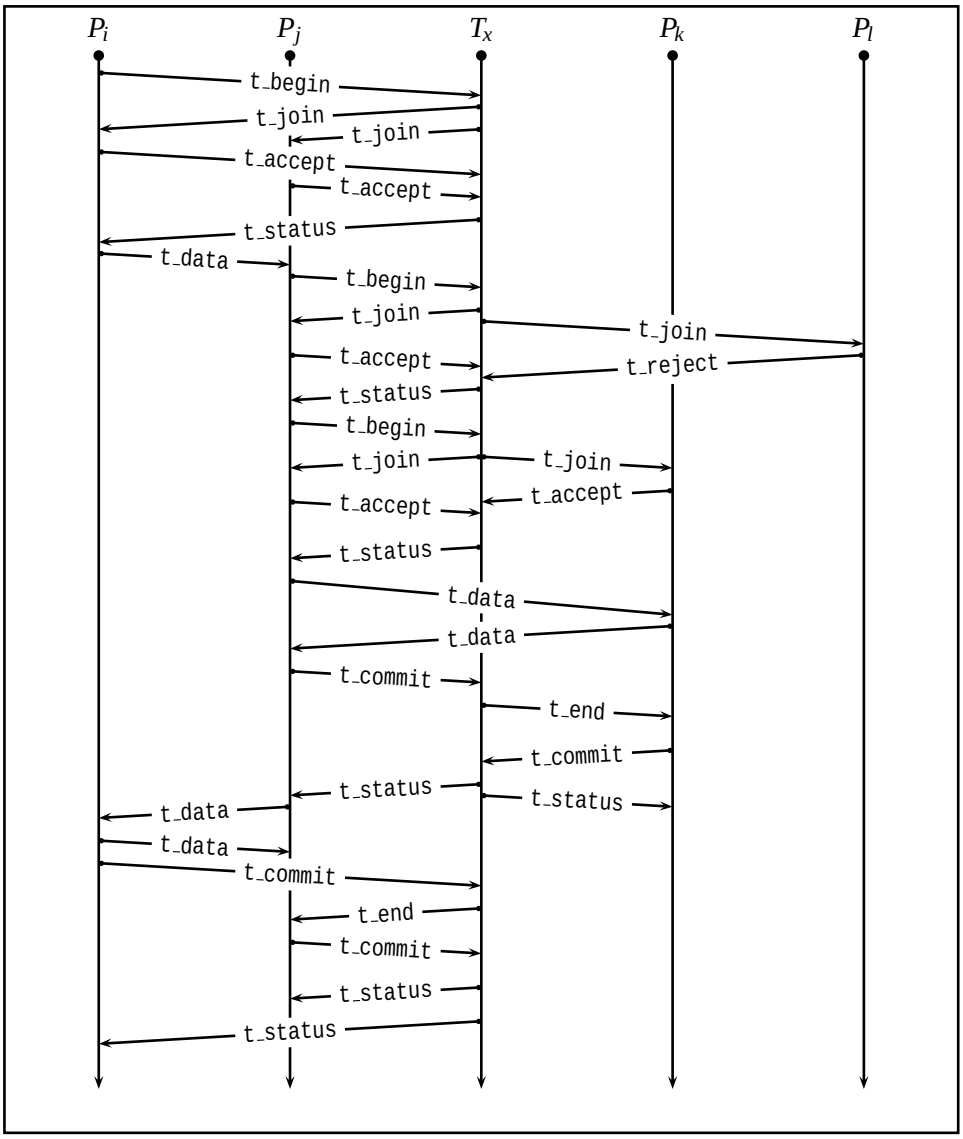

Figure 4.3: clio Transaction (4 Clients)

forming the echo subtransaction with the durability server. Also note that the durability server appears to the transaction server as just another client.

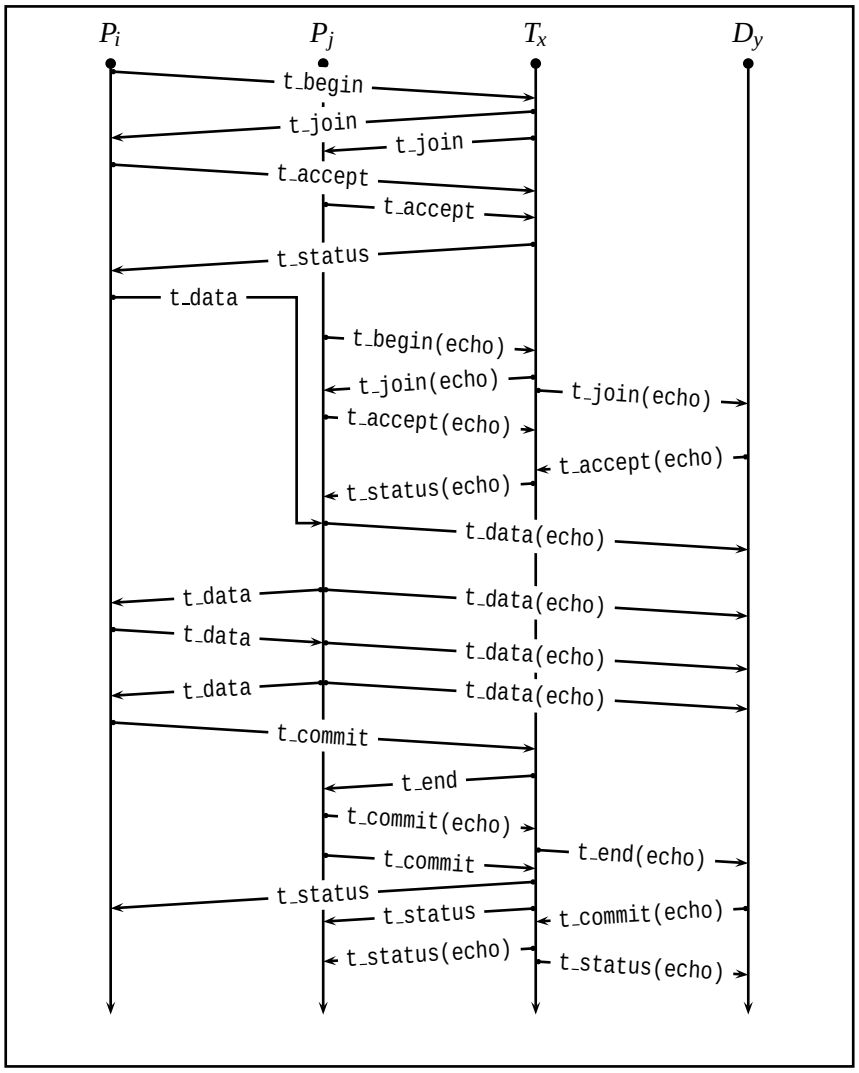

Figure 4.4: clio Echoed Subtransaction

4.3.5 Summary

clio supports:

agreement because of its use of a commit protocol.

consideration because the processes in a subtransaction compute the partial order of the subtransaction messages and commit to this order.

obligation because the transaction server does not confirm subtransaction message validity until and unless the parent transaction is committed, and commitment is done atomically; thus the obligations implied by the message are instantiated simultaneously.

durability because subtransaction data may be copied to the durability server and the durability server commitment is necessary for the transaction to be successful.

quantitative scalability because a transaction can have an unlimited number of equipollent subtransactions between independent processes.

qualitative scalability because subtransaction message order and data are arbitrary.

4.4 hermes

hermes, the son of Zeus and Maia, was the herald of the Olympian gods, and thus responsible for delivering messages securely. In Forum, it is the secure communication protocol, supporting the properties of privacy, integrity and origin (with and without non-repudiation). Uniqueness is not supported directly, as is discussed below. Section 4.4.1 describes the algorithms used for communication security and section 4.4.2 presents the design for hermes.

4.4.1 Algorithms

An overview of security is provided in section 2.3. Computer system security is the protection of the privacy, integrity and availability of information while it is stored on or communicated within the system [28, 9]. Computer security consists of mechanisms to insure that data can be disclosed, altered or made unavailable only if specifically authorized by the owner or controller of the data. This authorization is termed the security policy. In electronic commerce, the main focus is on communication security using cryptography to support the required properties.

The cryptographic operations can either be embedded in a separate protocol, providing a security layer that can be invoked by other protocols, or be provided as a library of functions to be used directly by those protocols which require security. In either case, it is necessary to define the security policy which is applicable for each message. These issues are discussed further in the following sections.

Cryptography

Originally, cryptography was used to make data unintelligible unless one possessed a secret or a key, thus supporting privacy[17]. More recently it has been used to support data integrity, origin and uniqueness.

Cryptography is accomplished using data transformations which are either invertible or non-invertible. An invertible transformation will have an encoding function and a corresponding decoding function, which convert data between plaintext and ciphertext. A non-invertible transformation will have only an encoding function, which converts plaintext to a digest. Keys are always used with invertible transformations; they may be used with non-invertible transformations; this is termed a signing function. Signing functions convert plaintext into a signature.

Cryptographic functions may use different algorithms which have varying strengths and computational requirements. The strength of encoding algorithms is based on the difficulty of decoding the ciphertext into the corresponding plaintext without the key. This is dependent mostly on the number of possible keys, the predictability of the plaintext and the algorithms used. The strength of hashing algorithms is based on the difficulty of specifying plaintext with a given digest or of specifying two plaintexts with the same digest.

Encoding and decoding functions may use symmetric or asymmetric algorithms. For symmetric algorithms, the same key is used to encode and decode data, and is termed a shared key. The main advantage of symmetric algorithms is performance, because these algorithms primarily use data substitutions which can be computed efficiently. The main

disadvantage is the lack of non-repudiation; because keys are shared, the signature used for origin could have been computed by either party.

For asymmetric algorithms, different keys are used to encode and decode data. One key, termed the private key, is known only by one party, while the other key, termed the public key, is known by all other parties. The main advantage of asymmetric algorithms is the ability to support non-repudiation; the separate keys make it possible to determine who constructed a given signature. The main disadvantage is performance since these algorithms are based on computationally intractable problems.

Security Protocols

For the security properties of privacy, integrity and origin, there are well-established security protocols [18, 29, 36]. The commonly used protocols will be defined in this section. Some of the protocols support multiple properties because the processing for one property may imply other properties.

Name	Purpose
`s`	the process's identifier (PID)
`r`	the other process's identifier
`K[]`	an array of shared keys indexed by PID
`Kpub[]`	an array of public keys indexed by PID
`Ks`	the private key of the process
`data`	the plaintext message
`enc_data`	the encoded message
`hash`	a message digest
`sign`	an encoded digest (signature)
`tmp`	temp variable

Table 4.1: Standard Variables

The security protocols are defined with the Abstract Protocol (AP) notation, a formal language that clarifies protocol definition and enables correctness proofs [12]. In AP,

a process is defined with a sequence of sections: inputs, constants, variables, functions and a set of guarded actions, as in [15]. During process execution, the set of action guards is repeatedly evaluated, and one of the actions whose guard is `true` is selected randomly and executed atomically. The logical disjunction of the guards must be `true`. Several variables and inputs are used throughout this section. These are defined in Table 4.1.

```
process s
input
  K : array[integer] of integer;
var
  data, enc_data, r : integer;
begin
  true → # sender
    data, r := random, random;
    enc_data := encode(data, K[r]);
    send(enc_data) to r;
□ rcv(enc_data) from r →
    data := decode(enc_data, K[r]);
end
```

Protocol 4.1: Symmetric Key Privacy

Privacy

To support privacy with a symmetric key algorithm, the sender encodes the plaintext message with a shared key. The receiver decodes the ciphertext message with the same key. (See Protocol 4.1.) Privacy may be correctly assumed if only the sender and receiver have the shared key.

Integrity and Origin

To support integrity with a symmetric key algorithm, the sender computes a digest of the message and encodes the digest with the shared key, producing a signature which is sent along with the message. The receiver decodes the signature with the shared key and compares it to the digest of the message; if they are equal, the message is accepted. (See Protocol 4.2.) Origin and integrity are coupled, because the digest is

```
process s
input
  K : array[integer] of integer;
var
  hash, sign, data, r : integer;
begin
  true → # sender
    data, r := random, random;
    hash := digest(data);
    sign := encode(hash, K[r]);
    send(data, sign) to r;
□ rcv(data, sign) from r →
    hash := decode(sign, K[r]);
    if hash ≠ digest(data) →
      # reject message
    □ hash = digest(data) →
      # accept message
    fi
end
```

Protocol 4.2: Symmetric Key Integrity and Origin

encoded with the shared key. Integrity and origin may be correctly assumed if only the sender and the receiver have the shared key.

To support non-repudiation as well as integrity with an asymmetric key algorithm, the sender computes a digest of the message and encodes the digest with its private key, producing a signature. The receiver computes a digest of the message, decodes the signature with the sender's public key and accepts the message if the computed digest matches the decoded signature. (See Protocol 4.3.) Integrity and non-repudiation may be correctly assumed if only the sender has the private key.

Uniqueness

To support uniqueness with a symmetric key algorithm, the sender computes a number (termed a nonce) which is both unpredictable and new to the current protocol execution; it then computes the digest of the concatenation of the message and the nonce, and encodes the digest with the shared key, producing a signature. The sender

```
process s
input
  Ks : integer;
  Kpub : array[integer] of integer;
var
  data, hash, sign, r: integer;
begin
  true → # sender
    data, r := random, random;
    hash := digest(data);
    sign := encode(hash, Ks);
    send(data, sign) to r;
□ rcv(data, sign) from r →
    hash := decode(sign, Kpub[r]);
    if hash ≠ digest(data) →
      # reject message
    □ hash = digest(data) →
      # accept message
    fi
end
```

Protocol 4.3: Asymmetric Key Integrity and Non-repudiation

includes the nonce and signature in the message.

The receiver checks the uniqueness of the nonce, and then compares the decoded signature with the digest of the concatenation of the message and the unique number. It accepts the message if the nonce is unique to this protocol execution and the signature is valid.

Uniqueness is not supported directly by hermes, because of the large storage and computational costs of verifying uniqueness. Instead, uniqueness is supported indirectly in Forum, because all (almost) clio messages are either themselves unique or are idempotent in their effect.

The t_data messages sent and received by clients are unique by design; each subtransaction identifier is unique, and each message within a subtransaction will have a unique sequence number because of the ack and prec fields in each message, since

each message will have different values for these.

The other messages, except one, are idempotent by design. These messages are sent between server and client processes and, when repeated, have no additional effect. This is readily seen for the `t_qstatus`, `t_qorder` and `t_qvalid` query and corresponding response messages, and for the `t_accept`, `t_reject`, `t_commit` and `t_abort` messages sent from clients to servers.

For the `t_join` and `t_end` messages sent from servers to clients, their repetition may eventually yield a different response from the client, but that response would have no effect since the original response would have already taken effect.

The one non-idempotent message is the `t_begin` message sent from clients to servers. These can be replayed since they are not unique, in and of themselves, and replaying them will cause the server to create a new subtransaction. It is up to the client software to detect the duplicate `t_join` message, perhaps using the `sbxdata` field, and to reject the spurious subtransaction.

Layered or Functional?

A major design issue in communication security is whether to have a separate protocol which provides the security properties for all other protocols (i.e. a security layer) or to have a library of functions for the basic security operations of encoding, decoding and hashing which can be invoked by other protocols to provide the security properties that pertain to that specific protocol.

There are significant advantages and disadvantages to each. The main advantages of layering are:

- a modular implementation, which leads to greater reliability;
- uniformity of user and administrative interfaces, which promotes ease of use and better key management;

- protocol-independent policy definition, which enables more complete policy enforcement.

The main disadvantages are that one size fits few, and the necessity of implementing the security protocol in the kernel so that it can be used for low-level network administration and virtual private networks. A security protocol is also significantly more difficult to design and implement. The main advantages of a functional approach are:

- the security can be tailored for the application, which enables a more precise policy definition;
- fewer political or organizational design compromises are required, which allows security to be implemented faster, cheaper and, perhaps, better.

The main disadvantages are that the different designs imply different administrative and user interfaces, especially for key management, and the total cost of security is increased because of design and implementation redundancies among the separate protocols.

Layering is certainly the better approach, particularly because of the more complete and uniform security policy support. And with proper design and careful programming, the disadvantages can be lessened or eliminated. Due to time constraints, however, the security properties will be supported with a library of functions which will be invoked by clio to secure each transaction message.

Communication Security Policy

All messages in a system do not require the same security properties. For example, if a process sends two messages to the same process, but one is routed over a physically secure network and the other is not, then only the second message may require privacy encoding by the sender. Or if the two messages are part of the same electronic commerce transaction, but the first is only a price inquiry and the second is an order, then only the second message will require non-repudiation.

The communication security policy determines which properties should be supported for a specific message. The policy may be based on message routing, sender or receiver identity, message security level or content, upper level communication protocol or various other system criteria. The policy is normally defined by the system administrators or programmers and the data owners.

Communication protocols can be security-neutral, in the sense that the protocol implies nothing about the security requirements of the data which is being transmitted with that protocol. FTP is a file transfer protocol which copies data files from one network node to another, without regard to the content of the data. On the other hand, protocols can directly imply which properties to apply to a specific message. SNMP is a network management protocol which transfers network control and status information between network nodes. All SNMP messages thus require both integrity and origin to insure the correct operation of the network.

In some cases, different parts of a message may require different properties. For example, a data transfer message in a distributed file system would contain a header with the storage address and size of the data, and the data itself. Only the latter of these would need to be encoded for privacy. Or an order message in an electronic commerce system would contain the name of the customer, the merchandise or service being ordered, and the price being paid. Public policy may prohibit or require the privacy of the customer name, based on whether firearms or medical services are being purchased.

4.4.2 Design

Two things will need to be specified by the design of hermes: the functions that clio can use for communication security, and the properties to be applied for each clio message.

Functions

The implementation of the hermes protocol was to include a security protocol and cryptographic and large integer libraries. Although the security protocol is not to be implemented, the two libraries were completed, and so these will be used to provide the following cryptographic functions:

sp_encode encodes a memory buffer with a symmetric key algorithm.

sp_decode decodes a memory buffer with a symmetric key algorithm.

sp_digest computes a digest for a memory buffer with a message digest algorithm.

sp_sign constructs a digital signature for a message buffer by computing a digest with sp_digest and then encoding that digest with an asymmetric key algorithm using the sender's private key.

sp_verify verifies a digital signature for a message buffer by computing a digest for the buffer with sp_digest and decoding the digital signature with an asymmetric key algorithm using the sender's public key, and then comparing the decoded signature to the digest.

To provide memory management, the following functions will be exported from the cryptographic library:

krypt_memget allocates a memory buffer, optionally erasing all data.

krypt_memput deallocates a memory buffer, optionally erasing all data.

krypt_memset fills a memory buffer with either a preset character or with random data.

krypt_memcpy copies a memory buffer from one location to another. The byte ordering of the data may be changed on a 16, 32 or 64 bit boundary. The buffers may overlap.

krypt_memcmp compares the contents of two memory buffers.

Message Properties

In general, the security properties required for a message depend upon how the data is to be used. The clio protocol needs particular attention because of the commercial nature of the protocol as well as the order protocol used for intra-client communication.

In order protocols each message is logically divided into two components: the **data** and an **ordinal**, which is used to compute the message order. The conjunction of the two in an order protocol message will be referred to as their **coincidence**. The security properties may be different for each of these.

Privacy

Privacy is not a requirement, in general, for order protocols, since the computation of the order relation does not depend upon the secrecy of any of the information; in fact, the opposite may be true.

But for a protocol used in electronic commerce, privacy is an obvious requirement for the data, and may be required for the ordinal and coincidence as well, since one party in a multi-party transaction may not want any other party to know about all subtransactions in which she is engaged.

hermes supports privacy for the `t_data` message *data* field and for the `t_begin` and `t_join` message *uid* and *sbxdata* fields. These fields define what is being transacted and between whom, and so they are encoded with a shared key to protect this information. Note that most of the other fields in clio messages are transaction and message identifiers, which imply little about the nature of the transaction beyond its complexity and size.

Integrity

The purpose of an order protocol is to compute a correct order relation, which is a set of tuples $\langle a, b \rangle$, such that b occurred after or because of a. The order protocol must rely on the integrity of the ordinal; otherwise, the first element of the tuple would

be incorrect; it must also rely on the data, which defines the nature of the event; otherwise the second element of the tuple would be incorrect.

Lastly, it must rely on the coincidence of ordinal and data; otherwise the conclusion would be incorrect. That is, an attacker could construct a message with a valid ordinal from one message and valid data from another message, which would cause the receiver of the message to compute an invalid causal relationship. Thus, order protocols require ordinal, data and coincidence integrity. And, in any event, the commercial usage of the protocol implies an integrity requirement for the data.

hermes supports integrity for all messages, by computing a message digest over the entire message. The digest is encoded with either a shared key or the sender's private key, depending on the type of origin required for the message. In both cases, the receiver can validate the integrity of the message and, for the `t_data` message between clients, the integrity of the message ordinal (the `ack` and `prec` fields) and data, and their coincidence.

Origin

The meaning of an ordinal is dependent on its origin - to say that it's 10:00 A.M. is meaningless without specifying the place at which the clock value is being read. Also, the validity of an ordinal may be dependent on its origin - process P_i attesting to the current value of the Lamport Clock at process P_j would be invalid.

Coincidence origin is required by definition - coincidence refers to the coupling of data and ordinal in one place or process. Coincidence origin, in turn, implies a requirement for data origin, and in any event, data origin is a strict requirement for electronic commerce.

These requirements could greatly complicate the calculation and validation of message origin, particularly for ordinal data which may originate in multiple processes. Fortunately, the design of clio considerably simplifies things, because the order data

is computed on a subtransaction basis, and each subtransaction involves only two processes. hermes supports origin by signing the encoded message digest calculated for the validation of integrity (described above).

Non-repudiation

It would seem at first glance that origin must be non-repudiable for all clio messages. This is not correct, however, because the only clio message which effectively transfers goods, service or money is the `t_commit` message. All other messages can be repudiable.

Interestingly, the message that might appear to require non-repudiation the most, the `t_data` message, is the one message that must be repudiable, because no obligation should be created until the subtransaction is deemed successful by the client with the transmission of the `t_commit` message.

hermes supports non-repudiation for the messages between the transaction server and the client which can affect the final status of the subtransaction – `t_accept`, `t_reject`, `t_commit` and `t_abort`. This should be sufficient to prove any legal claims that may arise from a transaction. The client encodes the digest of these messages with its private key, and the server saves these signatures for each successful transaction.

4.5 Summary

This chapter presents a new framework for electronic commerce. This framework supports many properties that are not part of existing frameworks. These consist of the contractual properties of **agreement**, **consideration**, **obligation** and **durability**, and the structural properties of **quantitative** and **qualitative scalability**. These properties are derived from contract law and existing commercial practices, and their inclusion in Forum enables the support of more general forms of electronic commerce.

Chapter 5

Implementation

5.1 Overview

Forum is a system for electronic commerce which consists of two components: HERMES, which supports the security properties required for electronic commerce, and CLIO, which supports the contractual and structural properties required for electronic commerce. This chapter describes the implementation of these two components according to the design presented in chapter 4.

The design for HERMES consists of a set of functions that protocols can invoke to provide their own communication security. The implementation uses this interface as a basis for a security function library, `libsp`. These security functions require cryptographic algorithms; these are provided in a cryptographic library, `libkrypt`, which also provides memory management functions. Asymmetric key encryption algorithms use large integer arithmetic operations; these are provided in a mathematics library, `libkmath`.

The design for CLIO consists of two servers, one to manage transactions and one to provide durability, and a protocol which enables clients to communicate with each other. The implementation includes a transaction server, `cliod`, and a durability server, `mnemd`, and a client library, `libxp`. Since much of the transaction and networking functionality is the same for server and clients, the common functions are implemented in a single library, `libclio`.

Figure 5.1 is a schematic of the Forum libraries and commands; the shaded sections indicate the HERMES libraries. Programming interfaces are indicated by dashed lines. The `libkrypt` box is angled up to indicate that it also provides memory management functions to `libclio`, `libxp` and the two servers. The implementations of the component libraries and servers are described in sections 5.2 and 5.3.

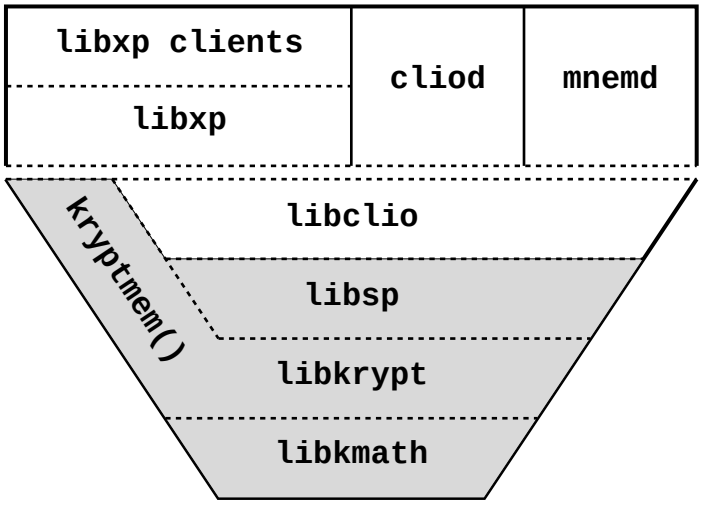

Figure 5.1: Forum Implementation

5.2 HERMES

The implementation of HERMES consists of three libraries, described in the following sections, starting from the bottom.

5.2.1 libkmath

DESCRIPTION

libkmath is a mathematics library that provides a set of mathematical functions for variable sized integers. These are useful for public key cryptography and other calculations which utilize very large integer values.

The original goal of libkmath was simply to augment libkrypt by providing support for those mathematical functions useful in cryptography. A more general approach was adopted, however, partly because the computation of the cryptographic functions require most of the arithmetic and logical operations anyway, and partly

because of the limited half-life of *ad hoc* software. Thus, completeness became a design goal for the library.

It also was decided fairly early to prefer portability to performance. Large integer libraries are often written for specific architectures, both to ease implementation and to enhance performance. These are worthwhile goals, but also lead to software with a short half-life. `libkmath` is implemented both to be independent of machine word size, byte order and instruction set and to support integers that are variable rather than fixed in length.

`libkmath` includes functions for variable creation and destruction, variable value assignment and retrieval, base and advanced arithmetic, logical comparisons, bit operations and cryptographic mathematics; for some of the arithmetic and logical operations, functions with one scalar operand are also included. And functions are provided to convert variables to and from binary, string and ASN.1 integer representations.

`libkmath` variables are accessed by means of opaque handles or descriptors. These handles are allocated and freed explicitly with `new()...free()` style calls. The library manages all memory allocations and deallocations for each variable, and automatically resizes variables to prevent overflow errors. The size of a `libkmath` integer is limited only by the virtual memory size of the underlying operating system.

An additional feature is the creation of temporary variables. For functions that return large integers, a temporary descriptor is allocated to hold the result if the main result variable is specified as a `NULL` descriptor, The temporary descriptor is returned as the result of the function, and will be freed after the next use.

TYPES

`libkmath` functions use the following non-standard types, which are also used by other Forum libraries and servers:

desc_t

is an opaque handle used to reference large integer operand and result variables. It is also used as the return value type for functions that return large integer results.

INT_size_t

is the **size_t** equivalent for libkmath variables, and thus limits their maximum value. This type is used in libkmath for size-related operands and results (for example, the power operand for bi_exp() and the return value of bi_log()). It is currently a 32 bit signed integer, and because it is also used as a bit address, the maximum size of a variable is 2^{28} (256MB).

boole

is used for boolean operands and results, and may have the value TRUE or FALSE or, when used as a result, DONTKNOW or BOOLERR, indicating an indeterminate result or an operational error, respectively. Thus, a non-zero value should not be construed as TRUE.

FUNCTIONS

Variable Creation, Assignment and Conversion

desc_t bi_new();

creates a new large integer variable with an initial value of 0, and returns the descriptor for that variable as the value of the function. This descriptor can be used in subsequent operations.

desc_t bi_set(desc_t *toD*, desc_t *fromD*);

assigns the value of the *fromD* operand to the *toD* parameter and as the result of the function.

desc_t bi_setint(desc_t *opD*, int *val*);

assigns the value of the *val* integer operand to the *opD* parameter and as the result of the function.

int bi_getint(desc_t *opD*, INT_size_t **magnitudep*);

returns the most significant bits (31 for a four byte integer) of the *opD* operand. If *opD* is NULL or invalid, the value 0 is returned. If *magnitudep* is not NULL, the magnitude of the variable (essentially the exponent base 2) is returned in this variable. A magnitude greater than zero implies some loss of precision in the returned value.

desc_t bi_sscan(desc_t *opD*, char **strval*, int *base*);

computes the value represented by the *strval* character string, and returns it in the *opD* parameter and as the result of the function. The *base* operand defines the base used for the string, ranging from two to sixteen, inclusive.

char *bi_sprint(char **bufp*, INT_size_t **bufsizep*, int *base*, desc_t *opD*);

returns a character string representation of the *opD* operand in the buffer pointed to by the *bufp* parameter and as the result of the function. The *bufsizep* operand specifies the maximum size of the buffer; upon successful completion, the actual length is returned; upon buffer overflow failure, the required length is returned. The *base* operand defines the base used for the string, ranging

from two to sixteen, inclusive.

desc_t bi_setasn1(desc_t *opD*, u_int8_t **bufp*, INT_size_t *bufsize*);

sets the value represented by the *bufp* ASN.1 octet string as the value of the *opD* parameter and as the result of the function. The ASN.1 octet buffer is of length *bufsize*. An ASN.1 value is a string of octets using two's-complement notation stored with the most significant octet first.

u_int8_t *bi_getasn1(u_int8_t **bufp*, INT_size_t **bufsizep*, desc_t *opD*);

returns an ASN.1 representation of the *opD* operand in the buffer pointed to by the *bufp* parameter and as the result of the function. The *bufsizep* operand specifies the maximum size of the buffer; upon successful completion, the actual length is returned; upon buffer overflow failure, the required length is returned.

void bi_free(desc_t *opD*);

deallocates the variable indicated by the *opD* operand.

Base Arithmetic Functions

desc_t bi_add(desc_t *sumD*, desc_t *augendD*, desc_t *addendD*);

sums the *addendD* and *augendD* operands and returns the result in the *sumD* parameter and as the result of the function.

desc_t bi_sub(desc_t *diffD*, desc_t *minuendD*, desc_t *subtrahendD*);

subtracts the *subtrahendD* operand from the *minuendD* operand and returns the result in the *diffD* parameter and as the result of the function.

desc_t bi_mul(desc_t *productD*, desc_t *multiplicandD*, desc_t *multiplierD*);

multiplies the *multiplicandD* operand with the *multiplierD* operand and returns the result in the *productD* parameter and as the result of the function.

desc_t bi_sqr(desc_t *squareD*, desc_t *opD*);

multiplies the *opD* operand with itself and returns the result in the *squareD* parameter and as the result of the function.

desc_t bi_div(desc_t *quotientD*, desc_t *remainderD*, desc_t *dividendD*, desc_t *divisorD*);

divides the *dividendD* operand by the *divisorD* operand and returns the result in the *quotientD* parameter and as the result of the function. If the *remainderD* parameter is not NULL, the remainder of the division is returned in this parameter. The *divisorD* operand must be non-zero.

desc_t bi_rem(desc_t *remainderD*, desc_t *dividendD*, desc_t *divisorD*);

divides the *dividendD* operand by the *divisorD* operand and returns the remainder in the *remainderD* parameter and as the result of the function. The *divisorD* operand must be non-zero.

desc_t bi_mod(desc_t *residueD*, desc_t *dividendD*, desc_t *modulusD*);

computes the residue of the *dividendD* operand divided by the *modulusD* operand and returns the result in the *residueD* parameter and as the result of the function. The *modulusD* operand must be positive.

Advanced Arithmetic Functions

desc_t bi_exp(desc_t *expD*, desc_t *baseD*, INT_size_t *power*);

computes the exponentiation of the *baseD* operand raised to the power specified by the *power* operand, and returns the result in the *expD* parameter and as the result of the function. The *power* operand must be non-negative.

INT_size_t bi_log(desc_t *remainderD*, desc_t *baseD*, desc_t *opD*);

computes the integer logarithm of the *opD* operand for the base specified by the *baseD* operand, and returns it as the result of the function. The value returned is the greatest value n such that the base raised to the n^{th} power is less than or equal to the exponent. If the *remainderD* parameter is not NULL, the difference between the value of *opD* and the value of *baseD* raised to the n^{th} power is returned in this parameter. The *baseD* and *opD* operands must be positive.

desc_t bi_sqrt(desc_t *sqrtD*, desc_t *remainderD*, desc_t *opD*);

computes the integer square root of the *opD* operand, and returns the result in the *sqrtD* parameter and as the result of the function. If the *remainderD* parameter is

not NULL, the difference between the value of *opD* and the square of the computed integer square root is returned in this parameter. The *opD* operand must be non-negative.

desc_t bi_mul_rnd(desc_t *rndD*, desc_t *opD*, desc_t *baseD*, int *how*);

computes the rounding of the *opD* operand to a multiple of the *baseD* operand, and returns the result in the *rndD* parameter and as the result of the function. The result is rounded up, down or to the closer multiple if the *how* operand is positive, negative or zero, respectively. The *baseD* operand must be positive.

desc_t bi_exp_rnd(desc_t *rndD*, desc_t *opD*, desc_t *baseD*, int *how*);

computes the rounding of the *opD* operand to a power of the *baseD* operand, and returns the result in the *rndD* parameter and as the result of the function. The result is rounded according to the *how* operand, as described above. The *baseD* operand must be positive, and the *opD* operand must be non-negative.

desc_t bi_fact(desc_t *factD*, INT_size_t *n*);

computes the factorial of the *n* operand, and returns the result in the *factD* parameter and as the result of the function. The *n* operand must be non-negative.

desc_t bi_fact_kofn(desc_t *factD*, INT_size_t *k*, INT_size_t *n*);

computes the factorial of the *n* operand divided by the factorial of the *k* operand, and returns the result in the

factD parameter and as the result of the function. This function implements the combinatorial selection *k of n* function. The *n* and *k* operands must be non-negative, and *k* must be less than or equal to *n*.

Logical Comparison Functions

boole bi_cmp_eq(desc_t *op0D*, desc_t *op1D*, boole *abs*);

compares the *op0D* and *op1D* operands, and returns TRUE if *opD* is equal to *op1D*, and FALSE otherwise. If *abs* is TRUE, the comparison is done using absolute operand values.

boole bi_cmp_ne(desc_t *op0D*, desc_t *op1D*, boole *abs*);

compares the *op0D* and *op1D* operands, and returns TRUE if *op0D* is not equal to *op1D*, and FALSE otherwise. If *abs* is TRUE, the comparison is done using absolute operand values.

boole bi_cmp_lt(desc_t *op0D*, desc_t *op1D*, boole *abs*);

compares the *op0D* and *op1D* operands, and returns TRUE if *op0D* is less than *op1D*, and FALSE otherwise. If *abs* is TRUE, the comparison is done using absolute operand values.

boole bi_cmp_le(desc_t *op0D*, desc_t *op1D*, boole *abs*);

compares the *op0D* and *op1D* operands, and returns TRUE if *op0D* is less than or equal to *op1D*, and FALSE otherwise. If *abs* is TRUE, the comparison is done using absolute operand values.

boole bi_cmp_gt(desc_t *op0D*, desc_t *op1D*, boole *abs*);

compares the *op0D* and *op1D* operands, and returns TRUE if *op0D* is greater than *op1D*, and FALSE otherwise. If *abs* is TRUE, the comparison is done using absolute operand values.

boole bi_cmp_ge(desc_t *op0D*, desc_t *op1D*, boole *abs*);

compares the *op0D* and *op1D* operands, and returns TRUE if *op0D* is greater than or equal to *op1D*, and FALSE otherwise. If *abs* is TRUE, the comparison is done using absolute operand values.

boole bi_cmp_zero(desc_t *opD*);

compares the *opD* operand to zero, and returns TRUE if *opD* is equal to zero, and FALSE otherwise.

boole bi_cmp_one(desc_t *opD*);

compares the *opD* operand to one, and returns TRUE if *opD* is equal to one, and FALSE otherwise.

boole bi_cmp_neg(desc_t *opD*);

compares the *opD* operand to zero, and returns TRUE if *opD* is less than zero, and FALSE otherwise.

boole bi_cmp_pos(desc_t *opD*);

compares the *opD* operand to zero, and returns TRUE if *opD* is greater than zero, and FALSE otherwise.

boole bi_cmp_nat(desc_t *opD*);

compares the *opD* operand to zero, and returns TRUE if *opD* is greater than or equal to zero, and FALSE otherwise.

boole bi_cmp_odd(desc_t *opD*);

checks the *opD* operand for indivisability by two and returns TRUE if *opD* is odd, and FALSE otherwise.

boole bi_cmp_even(desc_t *opD*);

checks the *opD* operand for divisability by two and returns TRUE if *opD* is even, and FALSE otherwise.

boole bi_cmp_prime(desc_t *opD*);

checks the *opD* operand for indivisibility by any number other than itself and one, and returns TRUE if *opD* is prime, and FALSE otherwise.

Bit Operation Functions

desc_t bi_and(desc_t *resultD*, desc_t *op0D*, desc_t *op1D*);

computes the logical conjunction of the *op0D* and *op1D* operands, and returns the result in the *resultD* parameter and as the result of the function.

desc_t bi_or(desc_t *resultD*, desc_t *op0D*, desc_t *op1D*);

computes the logical disjunction of the *op0D* and *op1D* operands, and returns the result in the *resultD* parameter and as the result of the function.

desc_t bi_xor(desc_t *resultD*, desc_t *op0D*, desc_t *op1D*);

computes the logical exclusive disjunction of the *op0D* and *op1D* operands, and returns the result in the *resultD* parameter and as the result of the function.

desc_t bi_nand(desc_t *resultD*, desc_t *op0D*, desc_t *op1D*);

computes the logical negation of the conjunction of the *op0D* and *op1D* operands, and returns the result in the *resultD* parameter and as the result of the function.

desc_t bi_nor(desc_t *resultD*, desc_t *op0D*, desc_t *op1D*);

computes the logical negation of the disjunction of the

op0D and *op1D* operands, and returns the result in the *resultD* parameter and as the result of the function.

desc_t bi_nxor(desc_t *resultD*, desc_t *op0D*, desc_t *op1D*);

computes the logical negation of the exclusive disjunction of the *op0D* and *op1D* operands, and returns the result in the *resultD* parameter and as the result of the function.

desc_t bi_not(desc_t *resultD*, desc_t *opD*);

computes the logical negation of the *opD* operand, and returns the result in the *resultD* parameter and as the result of the function.

desc_t bi_bitset(desc_t *resultD*, desc_t *opD*, INT_size_t *bit*);

sets a bit in the *opD* operand, and returns the result in the *resultD* parameter and as the result of the function. The *bit* operand specifies the number of the bit to be set (bit zero is the least significant bit).

desc_t bi_bitclr(desc_t *resultD*, desc_t *opD*, INT_size_t *bit*);

clears a bit in the *opD* operand, and returns the result in the *resultD* parameter and as the result of the function. The *bit* operand specifies the number of the bit to be cleared (bit zero is the least significant bit).

int bi_bittest(desc_t *opD*, INT_size_t *bit*);

tests a bit in the *opD* operand, and returns TRUE if the bit is set, and FALSE otherwise. The *bit* operand specifies the number of the bit to be tested (bit zero is the least significant bit).

desc_t bi_shift(desc_t *resultD*, desc_t *opD*, INT_size_t *shift*);

computes the logical shift of the *opD* operand, and returns the result in the *resultD* parameter and as the result of the function. The *shift* operand specifies the number of bits to be shifted; if *shift* is positive, the operand is shifted *up* (toward the more significant end); if the *shift* is negative, the operand is shifted *down* (toward the less significant end).

desc_t bi_rotate(desc_t *resultD*, desc_t *opD*, INT_size_t *rotate*);

computes the logical rotation of the *opD* operand, and returns the result in the *resultD* parameter and as the result of the function. The *rotate* operand specifies the number of bits to be rotated; if *rotate* is positive, the operand is rotated *up* (toward the more significant end); if *rotate* is negative, the operand is rotated *down* (toward the less significant end).

Cryptographic Mathematical Functions

desc_t bi_random(desc_t *ranD*, INT_size_t *bits*);

computes a random number of length *bits*, and returns the result in the *ranD* parameter and as the result of the function. The *bits* operand must be positive.

desc_t bi_prime(desc_t *primeD*, INT_size_t *bits*);

computes a prime number of length *bits*, and returns the result in the *primeD* parameter and as the result of the function. The *bits* operand must be positive.

desc_t bi_gcd(desc_t *gcdD*, desc_t *op0D*, desc_t *op1D*);

computes the greatest common divisor of the *op0D* and *op1D* operands, and returns the result in the *gcdD* parameter and as the result of the function. The *op0D* and *op1D* operands must be non-zero.

desc_t bi_lcm(desc_t *lcmD*, desc_t *op0D*, desc_t *op1D*);

computes the lowest common multiple of the *op0D*and *op1D* operands, and returns the result in the *lcmD* parameter and as the result of the function.

desc_t bi_mul_mod(desc_t *productD*, desc_t *multiplicandD*, desc_t *multiplierD*, desc_t *modulusD*);

computes the residue of the product of the *multiplicandD* and *multiplierD* operands divided by the *modulusD* operand, and returns the result in the *productD* parameter and as the result of the function. The *modulusD* operand must be positive.

desc_t bi_exp_mod(desc_t *expD*, desc_t *baseD*, desc_t *powerD*, desc_t *modulusD*);

computes the residue of the exponentiation of the *baseD* operand to the power specified by the *powerD* operand divided by the *modulusD* operand, and returns the result in the *expD* parameter and as the result of the function. The *modulusD* operand must be positive, and the *powerD* operand must be non-negative.

desc_t bi_inv_mod(desc_t *invD*, desc_t *opD*, desc_t *modulusD*);

computes the multiplicative modular inverse of the *opD* and

modulusD operands, and returns the result in the *invD* parameter and as the result of the function. The *modulusD* operand must be positive.

Scalar Functions

desc_t bi_scalar_add(desc_t *sumD*, desc_t *augendD*, int *addend*);

sums the *augendD* and *addend* operands and returns the result in the *sumD* parameter and as the result of the function.

desc_t bi_scalar_sub(desc_t *diffD*, desc_t *minuendD*, int *subtrahend*);

subtracts the *subtrahend* operand from the *minuendD* operand and returns the result in the *diffD* parameter and as the result of the function.

desc_t bi_scalar_mul(desc_t *productD*, desc_t *multiplicandD*, int *multiplier*);

multiplies the *multiplicandD* operand with the *multiplier* operand and returns the result in the *productD* parameter and as the result of the function.

desc_t bi_scalar_div(desc_t *quotientD*, desc_t *remainderD*, desc_t *dividendD*, int *divisor*);

divides the *dividendD* operand by the *divisor* operand and returns the result in the *quotientD* parameter and as the result of the function. If the *remainderD* parameter is not NULL, the remainder of the division is returned in this parameter. The *divisor* operand must be non-zero.

desc_t bi_scalar_rem(desc_t *remainderD*, desc_t *dividendD*, int *divisor*);

divides the *dividendD* operand by the *divisor* operand and returns the remainder of the division in the *remainderD* parameter and as the result of the function. The *divisor* operand must be non-zero.

desc_t bi_scalar_mod(desc_t *residueD*, desc_t *dividendD*, int *modulus*);

computes the residue of the *dividendD* operand divided by the *modulus* operand and returns the result in the *residueD* parameter and as the result of the function. The *modulus* operand must be positive.

desc_t bi_incr(desc_t *incr_opD*, desc_t *opD*);

increments the *opD* operand and returns the result in the *incr_opD* parameter and as the result of the function.

desc_t bi_decr(desc_t *decr_opD*, desc_t *opD*);

decrements the *opD* operand and returns the result in the *decr_opD* parameter and as the result of the function.

desc_t bi_abs(desc_t *abs_opD*, desc_t *opD*);

takes the absolute value of the *opD* operand and returns the result in the *abs_opD* parameter and as the result of the function.

desc_t bi_neg(desc_t *neg_opD*, desc_t *opD*);

takes the negation of the *opD* operand and returns the result in the *neg_opD* parameter and as the result of the function.

boole bi_scalar_cmp_eq(desc_t *op0D*, int *op1*, boole *abs*);

compares the *op0D* and *op1* operands, and returns TRUE if *opD* is equal to *op1*, and FALSE otherwise.

boole bi_scalar_cmp_ne(desc_t *op0D*, int *op1*, boole *abs*);

compares the *op0D* and *op1* operands, and returns TRUE if *op0D* is not equal to *op1*, and FALSE otherwise.

boole bi_scalar_cmp_lt(desc_t *op0D*, int *op1*, boole *abs*);

compares the *op0D* and *op1* operands, and returns TRUE if *op0D* is less than *op1*, and FALSE otherwise.

boole bi_scalar_cmp_le(desc_t *op0D*, int *op1*, boole *abs*);

compares the *op0D* and *op1* operands, and returns TRUE if *op0D* is less than or equal to *op1*, and FALSE otherwise.

boole bi_scalar_cmp_gt(desc_t *op0D*, int *op1*, boole *abs*);

compares the *op0D* and *op1* operands, and returns TRUE if *op0D* is greater than *op1*, and FALSE otherwise.

boole bi_scalar_cmp_ge(desc_t *op0D*, int *op1*, boole *abs*);

compares the *op0D* and *op1* operands, and returns TRUE if *op0D* is greater than or equal to *op1*, and FALSE otherwise.

EXAMPLES

```
#include <forum/kmath.h>
desc_t  myD;

myD = bi_new();
...
bi_free(myD);
```

A new libkmath variable is created with bi_new(); the new variable has a value of 0. After use, the variable is freed with bi_free().

```
#include <forum/kmath.h>
desc_t   myD;
char     *valstr = "12345678987654321234567890";
u_int8_t asn1str[] = {0x01};

bi_setint(myD, -9);         // sets myD to -9
bi_sscan(myD, valstr, 10);  // sets myD to valstr base 10
bi_setasn1(myD, asn1str, 1} // sets myD to 1
```

The value of the new variable may be assigned from either an integer, a string or, less commonly, an ASN.1 integer.

```
#include <forum/kmath.h>
desc_t   xD = bi_new(), yD = bi_new, zD = bi_new();

bi_setint(xD, 1);           // sets x to 1
bi_sscan(yD, "1", 10);      // sets y to 1
bi_add(zD, xD, yD);         // sets z to x+y (2)
bi_sqr(xD, zD);             // sets x to z^2 (4)
bi_sub(yD, zD, xD);         // sets y to z-x (-2)
bi_setint(xD, 1);           // sets x to 0x01
bi_setint(yD, 17);          // sets y to 0x11
bi_and(zD, xD, yD);         // sets z to (x & y) (0x01)
bi_xor(zD, xD, yD);         // sets z to (x ^ y) (0x10)
```

The value of the variable may also be set by being used as a result parameter in a libkmath arithmetic or bit function.

```
#include <forum/kmath.h>
desc_t   xD = bi_new(), yD = bi_new,
         zD = bi_new(), wD = bi_new();

bi_random(xD, 42);          // sets x to a 42 bit random num
bi_random(yD, 23);          // sets y to a 23 bit random num
bi_prime(zD, 56);           // sets z to a 56 bit prime num
bi_mul(wD, bi_mod(NULL, xD, yD), bi_exp(NULL, zD, 999));
                            // sets w to the product of
                            // x%y and z to the 999th power
```

This example shows the use of temp variables - both the divide and the exponentiation operation return a temp variable that will be deallocated when it is used for the multiply operation. (The random and prime numbers are exemplary, not purposeful.)

```
#include <forum/kmath.h>
desc_t   xD = bi_new(), yD = bi_new, zD = bi_new();

bi_setint(xD, 1);           // sets x to 1
bi_sscan(yD, "11", 16);     // sets y to 17 (11 base 16)
if (bi_cmp_gt(xD, yD, FALSE) == TRUE) {
    bi_set(zD, xD);         // if (x > y) z = x
}
```

Variable values can be compared using libkmath logical functions. Since the result of these functions can be an error indication, it is important to check explicitly for TRUE or FALSE. Most of these functions take an additional boolean parameter that denotes if the comparison should be made with absolute values.

```
#include <forum/kmath.h>
desc_t      myD = bi_new();
int         val;
char        valstr[3];
u_int8_t    asn1val[1];
INT_size_t  len1 = 1, len3 = 3;

bi_setint(myD, 42);                // sets myD to 42
intval = bi_getint(myD, NULL);     // sets intval to 42
bi_sprint(valstr, &len3, 11, myD); // sets valstr to "39"
bi_getasn1(asn1val, &len1, myD};   // sets asn1val to 0x2A
```

Finally, the value of a variable can be retrieved, either as an integer, a character string or an ASN.1 integer.

5.2.2 libkrypt

DESCRIPTION

libkrypt is a message codec library that provides functions to encode and decode data. The initial focus of the library is on cryptographic functions (hence the name), but the library interface is intended to allow the implementation of any data transformation, invertible or not.

The salient feature of libkrypt is its small, uniform interface for all algorithms. A process calls a krypt_new() function to obtain a descriptor for a named algorithm. and then calls krypt_setattr() and krypt_getattr() to configure the algorithm (setting keys for example).

After setting any needed attribute values, data is processed in either file mode or step mode. In file mode, the simpler of the two, the data is processed at once by

calling either krypt_stream() or krypt_buffer(), depending on whether the data is accessible via an I/O descriptor or is in memory.

In step mode, the data is processed one buffer at a time. The programmer calls krypt_start() to set the processing mode, and then calls krypt_step() one or more times. Step mode is normally used when the whole data stream is not available or does not fit into memory at the same time.

Lastly, when processing is completed the process calls krypt_free() to deallocate the descriptor and all resources associated with it. This can also be done at any time if an error occurs in processing.

With this generic interface, the programmer can encrypt and decrypt data with either a symmetrical or asymmetrical key algorithm, compute message digests, construct and verify signatures, obtain random data, translate character sets, encode and decode ASN.1 data, and compress and expand data (audio, video and text). The generic interface makes programming easier, makes the library more reliable because more of the code is shared between algorithms, and eases test case generation.

Algorithm attributes are a key part of the libkrypt programming interface. Attributes define different characteristics of each algorithm. Each libkrypt algorithm has a unique name, a type and a set of supported processing modes. It also may have an ASN.1 UUID and other type or algorithm specific attributes. Especially important are the step length, finish length and output length expansion attributes, which allow the programmer to calculate the necessary size for output buffers. Attributes are described further below.

libkrypt also provides memory management functions to allocate and free memory, and to set, copy and compare data values. Memory buffers are allocated with the krypt_memget() function and freed with the krypt_memput() function. The memory may be zeroed with either call. Memory buffers may be initialized with krypt_memset() to either a preset value or to random data. Memory buffers may

be copied with krypt_memcpy(), which also supports byte reordering on a 2, 4 or 8 byte boundary. Finally memory buffers may be compared with the krypt_memcmp() function.

TYPES

libkrypt functions use the following non-standard type, in addition to the desc_t and boole types used in libkmath.

kryptsize_t

is the **size_t** equivalent for libkrypt buffers. All input and output buffer size variables are of this type, and it is also used for many of the algorithm attributes. It is a signed integer which, in the current release, is 32 bits.

FUNCTIONS

Algorithm Initiation and Termination

int krypt_new_by_name(char *_name_, int *_countp_, desc_t *_kDp_);

looks up one or more algorithms by name. The *name* parameter specifies the name of the algorithm for which one or more descriptors are to be returned. The *countp* parameter is the address of an integer containing, at the start of the call, the maximum number of descriptors to be returned; on exit, this is the number of matching algorithms.

The *kDp* parameter points to an array of descriptors. If the value pointed to by **countp* is one, then only an exact match for the specified name is returned; if the value is greater than one, then any algorithm whose name contains

the specified name as a substring (minimum length of 3) will be returned, up to the number specified by the *countp* parameter.

desc_t krypt_new(char **name*);

is a simplified interface to krypt_new_by_name with an implicit count value of 1. The *name* parameter specifies the name of the algorithm for which a descriptor is to be returned. Only an exact match will be returned.

int krypt_new_by_uuid(u_int8_t **uuid*, int **countp*, desc_t **kDp*);

looks up algorithms by ASN.1 UUID. The *uuid* parameter specifies the ASCII string representation of the UUID to be matched. The other parameters are the same as for krypt_new_by_name().

int krypt_new_by_type(u_int16_t *type*, int **countp*, desc_t **kDp*);

looks up algorithms by type. The *type* parameter specifies the type to be matched. Types are defined for symmetric and asymmetric encryption, message digests, compression, random number generators, and alphabet and format conversion. The other parameters are the same as for krypt_new_by_name().

int krypt_free(desc_t *kD*);

deallocates an algorithm descriptor previously allocated by one of the krypt_new procedures, and all its associated resources. The *kD* parameter specifies the algorithm descriptor to free.

Algorithm Configuration and Status

int krypt_getattr(desc_t *kD*, u_int32_t *attr*, void **attr_valp*, kryptsize_t **attr_lenp*);

returns the current value for an attribute. The *kD* parameter specifies an algorithm descriptor returned from a previous call to one of the krypt_new() functions. The *attr* parameter specifies the attribute to be read and may be one of the standard attributes or may be an algorithm specific attribute. The *attr_valp* and *attr_lenp* parameters point to the attribute value buffer and length. For fixed length attributes, the value buffer length must be set exactly; for variable length attributes, the length must be at least long enough to hold the value. Upon completion, the actual length of the value is returned in this parameter.

int krypt_setattr(desc_t *kD*, u_int32_t *attr*, void **attr_valp*, kryptsize_t **attr_lenp*);

sets the value for an attribute. The *kD*, *attr*, *attr_valp* and *attr_lenp* parameters are as described for krypt_getattr(), except that the value buffer length must be set exactly.

Algorithm Execution

int krypt_start(desc_t *kD*, u_int16_t *mode*);

starts a new execution sequence. The *kD* parameter specifies an algorithm descriptor returned from a previous call to one of the krypt_new() functions. The *mode*

parameter specifies the mode for the new sequence (usually either encoding or decoding).

int krypt_step(desc_t *kD*, u_int8_t **input*, kryptsize_t *in_len*, u_int8_t **output*, kryptsize_t **out_lenp*, boole *finish*);

processes a single buffer of data according to the *mode* parameter to krypt_start(). The *input* and *in_len* parameters specify the input buffer address and length, and the *output* and *out_lenp* parameters specify the output buffer address and length address.

At the start of the call the value pointed to by *out_lenp* indicates the size of the output buffer; at the end of the call, it indicates the number of bytes written to the output buffer. Lastly, the *finish* parameter is set to TRUE to indicate the last step to be processed.

int krypt_stream(desc_t *kD*, u_int16_t *mode*, FILE **inf*, FILE **outf*);

processes a stream of data, and is equivalent to calling krypt_start() followed by one or more calls to krypt_step() with the *finish* parameter set to TRUE on the final call. The *kD* and *mode* parameters are the same as for the krypt_start() and krypt_step() functions. The *inf* and *outf* I/O descriptors are used as the data source and sink, respectively. The input stream must be open for reading, and the output stream must be open for writing. Data is read from the input stream until an EOF character is read.

int krypt_buffer(desc_t *kD*, u_int16_t *mode*, u_int8_t **input*, kryptsize_t *in_len*, u_int8_t **output*, kryptsize_t **out_lenp*);

processes a buffer of data in one step, similar to the krypt_stream() function. The parameters are the same as those used for krypt_start() and krypt_step(), without the *finish* parameter.

Memory Algorithms

void *krypt_memget(kryptsize_t *count*, boole *zero*);

allocates a new buffer. The *count* parameter specifies the buffer size in bytes, while the *zero* parameter specifies if the buffer is to be cleared upon allocation.

int krypt_memput(void **bufp*, boole *zero*);

deallocates a buffer previously allocated by krypt_memget(). The *bufp* parameter points to the buffer to be deallocated, while the *zero* parameter specifies if the buffer is to be cleared before deallocation.

kryptsize_t krypt_memcmp(void **xbufp*, void **ybufp*, kryptsize_t *count*);

compares two memory buffers pointed to by the *xbufp* and *ybufp* parameters. The number of bytes to compare is specified with the *count* parameter.

int krypt_memcpy(void **outp*, void **inp*, kryptsize_t *count*, int *size*, int *byteorder*);

copies data from one memory buffer to another, and does optional byte order conversions. The input and output

buffers are pointed to by the *inp* and *outp* parameters, respectively. The input and output buffers may overlap. The number of bytes to copy is specified with the *count* parameter, while the *size and byteorder* parameters specify the number of bytes per word and the byte ordering to be used for the output.

int krypt_memset(void *_bufp_, int _val_, kryptsize_t _count_);

initializes the buffer indicated by the *bufp* parameter. The *val* parameter specifies the initiation value which may be a number between zero and 255, or KRYPT_RANDOM, in which case the buffer will be initialized with random values. The *count* parameter specifies the number of bytes.

ATTRIBUTES

KRYPT_ATTR_NAME

The name of the algorithm, which is unique and is usually the common name of the algorithm or a particular implementation of it - e.g. IDEA, MD5, DES, KARN_DES.

KRYPT_ATTR_UUID

The ASN.1 identifier of the algorithm, if it has one. The UUID may not be unique within the library since the library may contain several versions of the same algorithm.

KRYPT_ATTR_TYPE

The type of the algorithm. This defines what the algorithm does, and may be one of the following values:

KRYPT_TYPE_HASH

message digest (MD2, MD4, MD5, SHA-1).

KRYPT_TYPE_SYMM

symmetric key encryption (DES, IDEA).

KRYPT_TYPE_ASYM

asymmetric key encryption (RSA, DSA).

KRYPT_TYPE_CMPRS

data compression (MPEG, JPEG).

KRYPT_TYPE_ECC

error detection, correction (HAMMING).

KRYPT_TYPE_RNG

random number generation (DEV_RANDOM).

KRYPT_TYPE_XLATE

data presentation, conversion (ASN.1).

KRYPT_ATTR_MODE

The modes supported by the algorithm, some of which may be algorithm or type specific. General modes are:

KRYPT_MODE_ENCODE

encodes, compresses, encrypts ...

KRYPT_MODE_DECODE

decodes, decompresses, decrypts ...

An example of a type specific mode is KRYPT_MODE_ECB (electronic code book), used for symmetric key block encoding algorithms.

KRYPT_ATTR_KEY

The cryptographic key used for encrypting or decrypting by keyed algorithms. Note that krypt_getattr() can be used to create a new key. The attribute KRYPT_ATTR_KEY_LEN defines the length of the key.

KRYPT_ATTR_STEP_LEN

The length of a processing step, in bytes. This is typically the block length of the algorithm. For example, for DES this value would be 8 and for MD5 this value would be 64.

KRYPT_ATTR_OUTPUT_LENX

The percentage by which the output length is expanded or contracted relative to the input length. For a symmetric cryptographic algorithm like DES or IDEA, this value would be 100, since the output length is equal to the input length. For a compression algorithm like MPEG, this value would be less than 100 for encoding and greater than 100 for decoding. For a message digest algorithm like MD2, this value would be 0 since these algorithms produce output only on the final step.

KRYPT_ATTR_FINISH_LEN

The additional length required in the output buffer on the final step. For a message digest algorithm, this value would be equal to the length of the digest (16 bytes for MD2).

EXAMPLES

```
#include <forum/krypt.h>
desc_t        md5D;
u_int8_t      *datap, *hashp;
kryptsize_t   datalen, steplen, hashlen, len = KRYPT_SIZE;
boole         finish = FALSE;
```

```
md5D = krypt_new("MD5");
krypt_start(md5D, KRYPT_MODE_ENCODE);
krypt_getattr(md5D, KRYPT_ATTR_STEP_LEN, &steplen, &len);
krypt_getattr(md5D, KRYPT_ATTR_FINISH_LEN, &hashlen, &len);
hashp = krypt_memget(finishlen, FALSE);
while (!finish) {
    finish = (datalen <= steplen);
    if (finish) steplen = datalen;
    krypt_step(md5D, datap, steplen, hashp, &hashhlen, finish);
    datalen -= steplen;
    datap += steplen;
}
```

The first example illustrates how to compute a message digest. The variables `data` and `data_len` are assumed to be initialized. The program calls `krypt_new()` with an argument of `"MD5"` to obtain a descriptor for the MD5 algorithm.

Next `krypt_start()` is called to begin encoding (the only mode supported by message digest algorithms) and then the step length and finish length attributes are read. These attributes are `KRYPT_SIZE` bytes in length. The step length specifies how many bytes MD5 expects in each step (except the last step), and the finish length of a message digest algorithm specifies how long the message digest will be.

After allocating a buffer for the digest, `krypt_step()` is called in a loop. Note that `finish` will be set to `TRUE` on the last call, which is when the digest will be written into the output buffer.

```
#include <forum/krypt.h>
desc_t        desD;
```

```
u_int8_t      *datap, *outputp, *keyp;
kryptsize_t   datalen, steplen, finishlen, outlen, outlenx,
              tmplen, lenlen = KRYPT_SIZE;
boole         finish = FALSE;

desD = krypt_new("DES");
krypt_set_attr(desD, KRYPT_ATTR_KEY, keyp, keylen);
krypt_start(desD, KRYPT_MODE_ENCODE);
krypt_getattr(desD, KRYPT_ATTR_STEP_LEN, &steplen, &lenlen);
krypt_getattr(desD, KRYPT_ATTR_FINISH_LEN, &finishlen, &lenlen);
krypt_getattr(desD, KRYPT_ATTR_OUTLENX_LEN, &outlenx, &lenlen);
outlen = ((datalen*outlenx)/100) + finishlen;
outputp = krypt_memget(outlen, FALSE);
while (!finish) {
    finish = (datalen <= steplen);
    if (finish) steplen = datalen;
    tmplen = outlen;
    krypt_step(desD, datap, steplen, outputp, &tmplen, finish);
    datalen -= steplen;
    datap += steplen;
    outputp += tmplen;
    outlen -= tmplen;
}
```

This example shows how to encode data with DES. The variables `datap`, `data_len`, `keyp` and `keylen` are assumed to be initialized. The processing is similar to that for MD5, except that the value of the key attribute is set and the value of the output length expansion attribute is read.

The KRYPT_ATTR_KEY attribute needs to be set for all symmetric key algorithms. The KRYPT_ATTR_OUTLENX attribute tells us how much bigger or smaller the output buffer needs to be as a percentage, compared to the input buffer length. A value of 100 would indicate equal lengths.

Note that the computation of the output buffer size is overdone a bit, since the output length expansion factor of DES is known to be 100. Also note that the finish length must be added because DES is a block algorithm, and so will pad the final block, or add another one if the data length is evenly divisible by the step length.

5.2.3 libsp

DESCRIPTION

libsp provides a set of functions for communications security, including:

- encoding and decoding data, with asymmetrical and symmetrical keys;
- computing and verifying data signatures, with or without repudiation;
- computing message digests.

The original goal of libsp was to support a general communication security protocol, but this has been scaled back to providing functions that communication protocols and applications can use for *ad hoc* security. The implementation of libsp is based directly on the design for hermes described in chapter 4.

TYPES

libsp functions use the previously defined types desc_t, boole and kryptsize_t.

FUNCTIONS

```
desc_t sp_new(u_int16_t mode, u_int8_t *skeyp, kryptsize_t
      skeylen, u_int8_t *akeyp, kryptsize_t akeylen, char
      *secretp);
```

creates a new security session descriptor. The *mode* parameter specifies the algorithms to be used, and should contain at most one algorithm of each type (message digest, symmetrical key and asymmetrical key). Currently supported algorithms include MD5, SHA-1, IDEA, DES and RSA.

The parameters *skeyp* and *skeylen* specify the address and length of the symmetrical key, and the parameters *akeyp* and *akeylen* specify the address and length of the asymmetrical key.

If the *secretp* parameter is NULL, the asymmetrical key is a public key which contains the identity of the other user in the security session; otherwise, the asymmetrical key is a private key which contains the identity of the invoking user, and the *secretp* parameter points to a character string which is hashed to produce a symmetric key that decrypts the private key.

A process will normally create a root security session with the private key of the user; this session will be used only to compute non-repudiable signatures and to decode data encoded with an asymmetric key algorithm.

int sp_digest(desc_t *spD*, u_int8_t **inp*, kryptsize_t *inlen*, u_int8_t **outp*, kryptsize_t **outlenp*);

computes a message digest in the security session specified by the parameter *spD*. The parameters *inp* and *inlen* specify the address and length of the input data. The parameter *outp* specifies the address of the message digest buffer, and the parameter *outlenp* specifies the address of

the length of this buffer, on input, and of the actual length of the digest, on successful completion.

int sp_encode(desc_t *spD*, boole *symm*, u_int8_t **inp*, kryptsize_t *inlen*, u_int8_t **outp*, kryptsize_t **outlenp*);

encodes data in the security session specified by the parameter *spD*. The parameter *symm* specifies whether to use symmetric or asymmetric key encryption. The parameters *inp* and *inlen* specify the address and length of the input data. The parameter *outp* specifies the address of the encoded data buffer, and the parameter *outlenp* specifies the address of the length of this buffer, on input, and of the actual length of the data, on successful completion.

Note that asymmetric key encryption uses the public key of the other user in the session as specified in the sp_new() function.

int sp_decode(desc_t *spD*, boole *symm*, u_int8_t **inp*, kryptsize_t *inlen*, u_int8_t **outp*, kryptsize_t **outlenp*);

decodes data in the security session specified by the parameter *spD*. The parameter *symm* specifies whether to use symmetric or asymmetric key decryption. The parameters *inp* and *inlen* specify the address and length of the input data. The parameter *outp* specifies the address of the decoded data buffer, and the parameter *outlenp* specifies the address of the length of this buffer, on input, and of the actual length of the data, on successful completion.

Note that asymmetric key decryption uses the private key of the user as specified in the sp_new() function, and so the descriptor specified in this case will be that of the root security session.

int sp_sign(desc_t *spD*, boole *repud*, u_int8_t **msgp*, kryptsize_t *msglen*, u_int8_t **signp*, kryptsize_t **signlenp*);

computes a signature in the security session specified by the parameter *spD*. The parameter *repud* specifies whether the signature is to be repudiable or not. The parameters *msgp* and *msglen* specify the address and length of the input data. The parameter *signp* specifies the address of the signature buffer, and the parameter *signlenp* specifies the address of the length of this buffer, on input, and of the actual length of the signature, on successful completion.

Note that a non-repudiable signature is generated with the private key of the invoking user as specified in the sp_new() function, and so the descriptor specified in this case will be that of the root security session.

int sp_verify(desc_t *spD*, boole *repud*, u_int8_t **msgp*, kryptsize_t *msglen*, u_int8_t **signp*, kryptsize_t *signlen*);

verifies a signature in the security session specified by the parameter *spD*. The parameter *repud* specifies whether the signature is repudiable or not. The parameters *msgp* and *msglen* specify the address and length of the input

data. The parameters *signp* and *signlen* specify the address and length of the signature buffer.

Note that a non-repudiable signature is verified with the public key of the other user in the session as specified in the sp_new() function.

void sp_free(desc_t *spD*);

deallocates the descriptor and all associated resources of the security session specified by the *spD* parameter.

kryptsize_t sp_digestlen(desc_t *spD*);

returns the message digest length for the security session specified by the *spD* parameter.

kryptsize_t sp_blocklen(desc_t *spD*);

returns the block length of the symmetric key algorithm for the security session specified by the *spD* parameter.

char *sp_userid(desc_t *spD*);

returns a pointer to the user identity associated with the asymmetric key for the security session specified by the *spD* parameter.

u_int8_t *sp_pubkey(desc_t *spD*, kryptsize_t **keylenp*);

returns a pointer to the public key of the user associated with the asymmetric key for the security session specified by the *spD* parameter. The *keylenp* parameter specifies the length of the returned key buffer.

kryptsize_t sp_encdatalen(desc_t *spD*, boole *symm*, kryptsize_t *datalen*);

returns the encoded data length for the security session specified by the *spD* parameter. The *datalen* parameter

specifies the input data length, and the *symm* parameter specifies whether to use symmetric or asymmetric key encryption.

kryptsize_t sp_signlen(desc_t *spD*, boole *repud*);

returns the signature length for the security session specified by the *spD* parameter. The parameter *repud* specifies whether the signature is to be repudiable or not.

EXAMPLES

```
#include <forum/krypt.h>
#include <forum/sp.h>
desc_t       secD, rootD;
u_int8_t     *myprivkeyp, *mysecretp, *pubkeyp, *skeyp;
kryptsize_t  mykeylen, pubkeylen, skeylen;

rootD = sp_new(0, NULL, 0, myprivkeyp, myprivlen, mysecretp);
secD = sp_new(0, skeyp, skeylen, pubkeyp, pubkeylen, NULL);
...
sp_free(secD);
sp_free(rootD);
```

This example illustrates initialization of the security library. First a root session descriptor is created with the user's private key and secret, then a normal session is created with the user identified by the *pubkeyp* variable. All key, length and secret variables are assumed to be initialized. Note that a security session created with a *mode* parameter of 0 will use the default algorithms (SHA-1, IDEA and RSA). Also note that the symmetric key parameters for the root security session are left unspecified - this session is normally only used to compute non-repudiable signatures.

```
#include <forum/krypt.h>
#include <forum/sp.h>
desc_t       secD = sp_new(...);
u_int8_t     *datap, *encdatap;
kryptsize_t  datalen, encdatalen;

encdatalen = sp_encdatalen(secD, datalen, TRUE);
encdatap = krypt_memget(encdatalen, 0);
sp_encode(secD, TRUE, datap, datalen, encdatap, &encdatalen);
...
krypt_memput(encdatap, 1);
sp_free(secD);
```

This example illustrates encrypting a data buffer. First the length of the encoded data buffer is computed and a buffer of that size is allocated. Next `sp_encode()` is called to encrypt the data using a symmetric key (the *symm* parameter is `TRUE`.) The `datap` and `datalen` variables are assumed to be initialized.

```
#include <forum/krypt.h>
#include <forum/sp.h>
desc_t       rootD = sp_new(...);
u_int8_t     *msgp, *signp;
kryptsize_t  msglen, signlen;

signlen = sp_signlen(rootD, TRUE);
signp = krypt_memget(signlen, 0);
sp_sign(rootD, FALSE, msgp, msglen, signp, &signlen);
if (!sp_verify(rootD, FALSE, msgp, msglen, signp, signlen)) {
```

```
    fprintf(stderr, "ERROR: verification failed\n");
}
krypt_memput(signp, 1);
sp_free(rootD);
```

In this example a message signature is computed and verified. First the signature length is computed and a buffer of that size is allocated. Then `sp_sign()` is called to sign the data using an asymmetric key (the *repud* parameter is `FALSE`). The signature is verified to be correct by calling `sp_verify()`. Note that the signature will be computed and verified with the user's own security session. The `msgp` and `msglen` variables are assumed to be initialized.

5.3 CLIO

The implementation of CLIO consists of two servers and a client programming library which can be used to implement CLIO applications. Central to all of these is the implementation of a protocol which enables clients to communicate with each other and the servers to create, conduct and terminate transactions. The implementation thus requires two kinds of functions: a set of lower level functions to send and receive the protocol messages, and a set of higher level functions to conduct and manage transactions.

At first glance, it might seem that the client and server[1] implementations of these functions would be disparate. Although both clients and servers send and receive clio protocol messages, almost all clio message types are exclusively sent by one and received by the other. For example, the message types `t_begin`, `t_accept` and `t_reject` are only sent by clients and only received by the transaction server, while the message types `t_join` and `t_end` are only sent by the transaction server and only received by clients. The only exception to this rule is the message type `t_data`, which one client sends to another.

[1] Note that the server being discussed is the transaction server; the durability server is, by design and implementation, a clio protocol client.

And there appears to be a similar bifurcation for the transaction layer. The transaction server creates and terminates transactions and subtransactions, while the clients compute the message order during the subtransaction. And afterwards, clients request information transaction status and message validity and ordering, which the server stores and provides upon request.

These appearances, however, turn out to be at least partly deceptive. The networking infrastructure – resolving names, sending and receiving messages, processing headers and security – is common to both clients and servers. And while each is the mirror image of the other in terms of messages transmitted and received, they both handle the same message types (with, again, `t_data` being the sole exception).

Lastly, most of the subtransaction and transaction data elements are common to both, to the point where a single data structure can be used for client and server subtransactions and transactions. This, in turn, allows common code to allocate, deallocate, enqueue, dequeue, initialize and update this data structure.

Given the amount of commonality, it seemed appropriate to use a shared code base for client and servers. `libclio`, which contains the common functions, is discussed in section 5.3.1. This library is used as the basis for the CLIO client library, `libxp`, the transaction server, `cliod`, and the durability server, `mnemd`.[2] These are described in sections 5.3.2, 5.3.3 and 5.3.4.

5.3.1 `libclio`

DESCRIPTION

`libclio` is intended to serve as a common basis for the implementation of CLIO server programs and client libraries. The implementation of `libclio` consists primarily of a transaction layer, a messaging layer, and a secure networking substructure.

[2] After Mnemosyne, the mother of the Muses (by Zeus), including Clio.

The secure networking substructure has two components. The data transmission component constructs a clio protocol header for a message, and sends the message with the security properties of origin and integrity. The data reception component polls active connections for new data and, when data is received, parses the message header and verifies the origin and integrity of the message.

The libclio messaging layer contains synchronous and asynchronous functions which correspond directly to each clio protocol message type. For example, there are `clio_snd_qstate()` and `clio_rcv_qstate()` functions for the message type `CLIO_MSG_T_QSTATE`.

The synchronous message functions send a clio protocol message, taking as arguments the data components of the message. The function will allocate space for a message and add the data to the message. The data is encrypted for privacy if the message type requires it. The message is then passed to the lower layer for transmission.

The asynchronous message functions receive and process clio protocol messages. After the lower level data reception component gets a message, it reads the message type and invokes the corresponding message receive function. This function will parse the message data fields into components, decrypting them if necessary.

The libclio transaction layer also contains synchronous and asynchronous functions which correspond to each libclio protocol message type. For example there are `clio_sbx_commit()` and `clio_sbx_committed()` functions for the message type `CLIO_MSG_T_COMMIT`. Each function operates on a specified subtransaction.

The synchronous functions compute the data component arguments for the corresponding message transmission function, and then invoke that function. For example, `clio_sbx_commit()` computes the final subtransaction order which is passed to the `clio_snd_commit()`. The attributes of the specified subtransaction may also be changed.

The asynchronous functions are invoked by the message reception function, passing as arguments the parsed components from the message data. The function then will update the specified subtransaction, and may raise an event to be handled by the client library or server program which is using libclio.

The programming model for libclio consists of the synchronous transaction layer functions and a set of asynchronous events coupled with an event handler. A libclio process begins by invoking a start routine which will establish the process's identity and security data, initialize the transaction and networking layers, and then enter a polling loop waiting for incoming data or connection requests. When data is received, it is processed by the two layers of libclio, and an event may be raised as well, depending on the particular message type.

At a specified polling interval libclio will call the event handler with a pseudo-event to provide an opportunity for its user (the library or server program) to perform administrative tasks, handle errors or interface with a real user. Note that while the event handler is executing, it may invoke the synchronous transaction layer functions to initiate or terminate subtransactions, or to send data or status requests.

The libclio message and data types along with the transaction and subtransaction states are described next. Note that the functions and events of the libclio programming model will not be specified here, because of their similarity to the functions and events of the libxp programming model; libxp is intended as a general client programming interface and so is covered in greater detail in section 5.3.2.

MESSAGES

CLIO_MSG_T_BEGIN is sent by a client to a transaction server to initiate a new subtransaction.

CLIO_MSG_T_JOIN is sent by a transaction server to a client to

invite the client to join a new subtransaction.

CLIO_MSG_T_ACCEPT is sent by a client to a transaction server to accept a new subtransaction.

CLIO_MSG_T_REJECT is sent by a client to a transaction server to reject a new subtransaction.

CLIO_MSG_T_DATA is sent by a client to another client to transmit subtransaction data.

CLIO_MSG_T_END is sent by a transaction server to a client to end a subtransaction.

CLIO_MSG_T_COMMIT is sent by a client to a transaction server to commit a subtransaction.

CLIO_MSG_T_ABORT is sent by a client to a transaction server to abort a subtransaction.

CLIO_MSG_T_QSTATE is sent by a client to a transaction server to request (sub)transaction state.

CLIO_MSG_T_STATE is sent by a transaction server to a client to report (sub)transaction state.

CLIO_MSG_T_QVALID is sent by a client to a transaction server to verify message validity.

CLIO_MSG_T_VALID is sent by a transaction server to a client to report message validity.

CLIO_MSG_T_QORDER is sent by a client to a transaction server to verify message order.

CLIO_MSG_T_ORDER is sent by a transaction server to a client to report message order.

CLIO_MSG_T_QECHO is sent by a client to a durability server to request archived message data.

CLIO_MSG_T_ECHO is sent by a durability server to a client to report archived message data.

TYPES

libclio functions use the following non-standard types, in addition to the previously defined types desc_t, boole and kryptsize_t.

clio_sbxid_t

a scaler type which specifies the conjoined transaction identifier and subtransaction number. The current implementation is a 64 bit unsigned integer, with 16 bits used for the subtransaction number.

STATES

CLIO_SBXSTATE_BEGINNINGindicates the subtransaction or transaction is initializing.

CLIO_SBXSTATE_ACCEPTINGindicates that one client has accepted the subtransaction.

CLIO_SBXSTATE_VOIDindicates one or both clients rejected the subtransaction; represents a terminal state.

CLIO_SBXSTATE_ACTIVEfor a subtransaction, indicates both clients accepted the join request and that the subtransaction is currently active; for a transaction, indicates that at least one subtransaction is currently active.

CLIO_SBXSTATE_COMMITTING indicates that one client has voted to commit the subtransaction.

CLIO_SBXSTATE_ABORTED for a subtransaction, indicates one or both clients voted to abort the subtransaction, or that their committed message orders did not agree; for a transaction, indicates that at least one subtransaction was aborted; represents a terminal state.

CLIO_SBXSTATE_COMMITTED for a subtransaction, indicates both clients voted to commit the subtransaction and agreed on the partial order of subtransaction messages; for a transaction, indicates that all subtransactions were either committed or voided; represents a terminal state.

5.3.2 libxp

DESCRIPTION

libxp provides programming interfaces for CLIO client processes to conduct electronic transactions. Functions are provided to:

- create subtransactions within new or existing transactions;
- to join or reject an invitation to participate in subtransactions;
- to transmit and receive data within a subtransaction and, optionally, to archive the subtransaction data;
- to conclude subtransaction by aborting or committing;
- to obtain status information about transactions and subtransactions, including the ability to verify subtransaction message integrity and order;
- to recover archived data from successful transactions.

The programming model consists of a set of functions that are called synchronously and a set of events that are delivered asynchronously to the client's event handler.

The event handler, along with the process's user identity, private key and public key, are defined in the libxp initialization function.

The libxp initialization function returns only in the event of an error. The client does its work when its event handler is run. The event handler is invoked when:

- the client receives a request to join or end a subtransaction;
- the client receives data from another client or status information from a transaction server;
- periodically, at a specified polling interval, so that the client can do administrative tasks or participate actively in other subtransactions.

The events delivered to the event handler and the event specific data are defined below under **EVENTS** and **TYPES**.

To create a new subtransaction, the client specifies the name of the other party and the name of the transaction server. libxp communicates with the transaction server which will then send a join request to both clients. If both accept[3] the subtransaction is formed.

The two clients in the subtransaction may then exchange data messages; the partial order of these is computed by libxp. The subtransaction is concluded when one client sends a commit or abort request to the transaction server. If a commit request is sent, libxp includes the computed partial order. The transaction server will send an end request to the other client, which then will also either commit or abort.

If both clients commit, and their message partial orders match, then the subtransaction is successful. If all accepted subtransactions in a transaction are successful, then the transaction is successful. For successful transactions, clients may request the validity and order of subtransaction messages from the transaction server. The transaction server will also report the status of transactions and subtransactions.

[3] libxp accepts automatically for the creator; the join request specifies the new subtransaction number.

If the client wishes to save a copy of subtransaction data, it specifies a durability server during subtransaction initiation. libxp will initiate an adjunct subtransaction with the durability server, and 'echo' all of the main subtransaction data to the durability server. libxp will also automatically commit (or abort) the adjunct subtransaction when the main subtransaction is committed (or aborted).

libclio provides much of the functionality of libxp. Specific to libxp are:

- named subtransactions; in libclio subtransactions are known only by their numeric subtransaction identifier, so libxp provides a name to number mapping;
- message identifier storage; in libclio messages are known only by message identifier, a 160-256 bit number; libxp retains and maps the message identifier to the sent and received message sequence number, which allows a user to reference a message as the n^{th} message sent or received rather than as message $0x823B424893\ldots C493EF23A2$;
- constructing and parsing client specific messages;
- computing client subtransaction and transaction attribute values and states;
- client order computation.

In the end, libclio constituted approximately 60% of the lines of code in libxp.

EVENTS

XP_EVENT_JOIN indicates the client is being invited to join a new subtransaction with another client by a transaction server.

XP_EVENT_DATA indicates the client has received subtransaction data.

XP_EVENT_END indicates the other client wishes to terminate a subtransaction.

XP_EVENT_STATE indicates transaction or subtransaction status has been received from the transaction server.

XP_EVENT_VALID indicates a response from the transaction server for a message validity query.

XP_EVENT_ORDER indicates a response from the transaction server for a message order query.

XP_EVENT_POLL is a pseudo-event with which libxp invokes the event handler at the requested polling interval.

TYPES

libxp functions use the following non-standard types, in addition to the previously defined types desc_t, boole and kryptsize_t.

xp_event_join_t

the event specific data for XP_EVENT_JOIN. It is a structure that includes the following members (note that the first two are input to the event handler, and the last two are output):

```
char  *xej_other;          // name of other client
char  *xej_xsrvr;          // name of transaction srvr
char  xej_name[];          // name of subtransaction
char  xej_dsrvr[];         // name of durability srvr
```

xp_event_data_t

the event specific data for XP_EVENT_DATA. It is a structure that includes the following members:

```
u_int8_t    *xed_datap;     // received data address
kryptsize_t xed_datalen;    // received data size
```

xp_event_state_t

the event specific data for XP_EVENT_STATE. It is a structure that includes the following member:

```
u_int8_t   xes_state;       // received state
```

xp_event_valid_t

the event specific data for XP_EVENT_VALID. It is a structure that includes the following members:

```
kryptsize_t xev_msgnum;    // number of message
boole       xev_sent;      // sent or rcvd
boole       xev_isvalid;   // valid or not
```

xp_event_order_t

the event specific data for XP_EVENT_ORDER. It is a structure that includes the following members:

```
kryptsize_t xeo_msgnum0;    // number of 1st message
boole       xeo_sent0;      // sent or rcvd
kryptsize_t xeo_msgnum1;    // number of 2nd message
boole       xeo_sent1;      // sent or rcvd
int         xeo_order;      // message order
```

xp_evnthdlr_t

the procedure type for a libxp event handler. The handler takes three arguments: the type of the event, the libxp descriptor of the affected subtransaction, and a void pointer (which will actually point to the type specific data defined above). The function returns a boole, which is only used for the XP_EVENT_JOIN, XP_EVENT_END and XP_EVENT_POLL events. For the first two, TRUE implies join

or commit, and FALSE implies reject or abort, respectively. For the XP_EVENT_POLL event, FALSE indicates that transaction processing is to be terminated. The type is defined as:

```
typedef boole (*xp_evnthdlr_t)(u_int8_t, desc_t, void *);
```

FUNCTIONS

int xp_start(u_int8_t *_privkeyp_, kryptsize_t _privkey_len_, char *_secretp_, xp_hndlr_t *_hndlr_, int _poll_);

initializes the libxp subsystem for the caller. The *privkeyp* and *privkey_len* parameters specify the address and length of the buffer which contains the user's private key; the private key also includes the user's name and public key. The *secretp* parameter specifies the address of a character string that contains the user's secret, which is used to decode the private key.

The *hndlr* parameter specifies the name of a libxp event handler, which will be invoked when an event occurs, and the *poll* parameter specifies the polling interval. This function will return immediately if an initialization error occurs; otherwise it returns after all transactions are completed.

desc_t xp_new(char *_name_, char *_other_, char *_xsrvr_, char *_dsrvr_, char *_sibling_);

initiates a new subtransaction in either an existing or a new transaction. The *name* parameter is the name of the new subtransaction. The *other* parameter is the name of the

other client. The *xsrvr* parameter is the name of the transaction server. The *dsrvr* parameter is the name of the durability server or NULL if archiving is not needed. The *sibling* parameter is the name of a sibling subtransaction; if not NULL the new subtransaction is created in the sibling's parent transaction; if NULL, the new subtransaction will be created in a new transaction.

A CLIO_MSG_T_BEGIN message is sent to the transaction manager, which will send CLIO_MSG_T_JOIN messages to both clients. The join request will be automatically accepted by the creator of the subtransaction; the other client's event handler will get an XP_EVENT_JOIN event, for which the xej_other datum will specify the name of the creator, and the xej_xsrvr datum will specify the name of the transaction server.

The event handler returns TRUE to accept the subtransaction and FALSE to reject it. If the subtransaction is accepted, then the event handler returns the name of the new subtransaction in the xej_name datum, and (optionally) the name of the durability server in the xej_dsrvr datum.

int xp_send(desc_t *xpsbxD*, u_int8_t **datap*, kryptsize_t *data_len*);

sends a CLIO_MSG_T_DATA message to the other client in a subtransaction. The *xpsbxD* parameter specifies the descriptor of the subtransaction. The *datap* and *data_len* parameters specify the address and length of the data buffer to be sent.

When the message is received, the other client's event handler will get an XP_EVENT_DATA event, with the xed_datap and the xed_datalen data specifying the address and length of the transmitted data.

int xp_commit(desc_t *xpsbxD*);

sends a CLIO_MSG_T_COMMIT message to the transaction server. This signals the client's intention to conclude the transaction successfully. The *xpsbxD* parameter specifies the descriptor of the subtransaction.

The server will send a CLIO_MSG_T_END message to the other client. When the message is received , the other client's event handler will get an XP_EVENT_END event (with no event specific data). The event handler returns TRUE to commit the subtransaction and FALSE to abort it.

int xp_abort(desc_t *xpsbxD*);

sends a CLIO_MSG_T_COMMIT message to the transaction server. This signals the client's intention to conclude the transaction unsuccessfully. The *xpsbxD* parameter specifies the descriptor of the subtransaction.

The server will send a CLIO_MSG_T_END request to the other client to determine its intention; this is processed as described above, but in any event the subtransaction will be aborted.

int xp_qstate(desc_t *xpsbxD*);

sends a CLIO_MSG_T_QSTATE message to the transaction server. The *xpsbxD* parameter specifies the descriptor of the subtransaction whose state is to be queried.

If the query is successful, the transaction server returns a CLIO_MSG_T_STATE message to the client; when the message is received, the client's event handler will get an XP_EVENT_STATE event. The xes_state datum will be one of the states listed in section 5.3.1.

int xp_qvalid(desc_t *xpsbxD*, kryptsize_t *msgnum*, boole *sent*);

sends a CLIO_MSG_T_QVALID message to the transaction server. The *xpsbxD* parameter specifies the descriptor of the subtransaction for which message validity is to be queried. The *msgnum* and *sent* parameters identify the message. (If *msgnum* is n and *sent* is TRUE, then validity would be determined for the n^{th} sent message.)

If the query is successful, the transaction server returns a CLIO_MSG_T_VALID message to the client; when the message is received, the client's event handler gets an XP_EVENT_VALID event. The xev_valid datum will be TRUE if the message is valid, and FALSE if it is invalid or the subtransaction was unsuccessful, and DONTKNOW if the subtransaction hasn't terminated.

int xp_qorder(desc_t *xpsbxD*, kryptsize_t *msg0num*, boole *sent0*, kryptsize_t *msg1num*, boole *sent1*);

sends a CLIO_MSG_T_QORDER message to the transaction server. The *xpsbxD* parameter specifies the descriptor of the subtransaction for which message order is to be queried. The *msg0num* and *sent0* parameters identify the first message, and the *msg1num* and *sent1* parameters identify

the second message.

If the query is successful, the transaction server returns a CLIO_MSG_T_ORDER message to the client; when the message is received, the client's event handler will get an XP_EVENT_ORDER event. The xeo_order datum will be 1 if the first message occurred after the second message, -1 if before, and 0 if they were simultaneous.

char *xp_name(desc_t *xpsbxD*);

returns a pointer to the name of the subtransaction specified by the *xpsbxD* parameter.

desc_t xp_lookup(char **name*);

returns a descriptor for the subtransaction specified by the *name* parameter.

EXAMPLES

```
#include <forum/xp.h>
#include <forum/krypt.h>

u_int8_t       *secretp, *privkeyp;
kryptsize_t    privkey_len;
xp_hndlr_t     my_hndlr;
if (my_init()) {
   xp_start(privkeyp, privkey_len, secretp, my_hndlr, 42);
}
exit();
```

This is a (more or less) complete main procedure for the typical libxp client. After the client-specific initialization, the client calls xp_start(). All other client process-

ing is done in the event handler. The private key and secret variables are assumed to be initialized.

```
#include <forum/xp.h>
#include <forum/krypt.h>
boole
my_hndlr(u_int8_t event, desc_t xpsbxD, void *evntdata) {
   boole  rc = TRUE;
   switch(event) {
      case XP_EVENT_JOIN : rc = join_hndlr(evntdata);
      break;
      case XP_EVENT_DATA : data_hndlr(xpsbxD, evntdata);
      break;
      case XP_EVENT_END : rc = end_hndlr(xpsbxD);
      break;
      case XP_EVENT_POLL : rc = poll_hndlr();
      break;
      default : rc = FALSE;
      break;
   }
   return(rc);
}
```

This is a simple libxp event handler. It simply switches on the event type and calls the appropriate handler. Note that subtransaction descriptor isn't passed to the join handler because, at this point, there isn't a subtransaction to reference. Also note that the end handler doesn't receive the event specific data, because there isn't any for this event.

```
#include <forum/xp.h>
#include <forum/krypt.h>
boole
join_hndlr(desc_t xpsbxD, xp_event_join_t *xejp) {
   boole  rc = FALSE;
   char   accept = 'n',
          *other = xejp->xej_other,
          *xsrvr = xejp->xej_xsrvr);
   fprintf(stdout, "JOIN: from %s server %s\n", other, xsrvr);
   fprintf(stdout, "accept? y/n sbxname ");
   fscanf(stdin, "%c %s\n", &accept, xejp->xej_name);
   if (accept == 'y')
      rc = TRUE;
   return(rc);
}
```

This routine handles join requests. It prints a message for the user indicating the nature of the request, the name of the other client and the name of the server. It then asks the user to accept (and name) or reject the new subtransaction. If the subtransaction is accepted. the routine returns `TRUE`. Note the implicit cast of the event data to the type appropriate for this event.

```
#include <forum/xp.h>
#include <forum/krypt.h>
void
data_hndlr(desc_t xpsbxD, xp_event_data_t *xedp) {
   char   *sbxname = xp_name(xpsbxD),
          *datap = xedp->xed_datap;;
```

```
    fprintf(stdout, "DATA: sbx %s:: %s\n", sbxname, datap);
    return;
}
```

This routine handles received data. It prints a message for the user indicating the event, the name of the subtransaction and the data (assumed to be a C character string).

5.3.3 cliod

`cliod` is the transaction server in Forum; it manages transaction initiation and termination, and also provides transaction state and message order and validity information to clients. The current version of the transaction server is entirely on-line, that is, it starts from scratch each time it is run. While this limits its usefulness, it is sufficient for its purpose as a proof of concept, and it does make the command interface much easier. `cliod` only needs to know its private key, and the secret which decodes it.

The `cliod` command accepts a single argument, the name of the file which contains its private key:

`cliod -k` *`keyfile`*

After reading the contents of the *keyfile*, `cliod` will prompt the user for the secret. It runs as a daemon after that.

Much of the functionality for `cliod` is provided by `libclio`. The server specific tasks were:

- constructing and parsing server specific messages;
- computing server subtransaction and transaction attribute values and states;
- storing and reporting subtransaction state and data, most importantly the subtransaction order information.

In the end, `libclio` provided approximately 75% of the total lines of code in `cliod`.

5.3.4 mnemd

`mnemd` is the durability server in Forum; it archives and reports subtransaction data when requested. The current version of the durability server retains subtransaction data across restarts, but at this point only provides the durability gained by 'sync-ing' the filesystem.

The `mnemd` command accepts a single argument, the name of the file which contains its private key:

```
mnemd -k keyfile
```

After reading the contents of the *keyfile*, `mnemd` will prompt the user for the secret. It runs as a daemon after that.

Worth noting is that although `mnemd` could have been implemented as a proper `libxp` client, it was not, since it required its own naming scheme and didn't require the message identifier storage and mapping of `libxp`.

`mnemd` does, however, utilize the client message and transaction functionality of `libxp` and, of course, `libclio`. It also requires:

- storing subtransaction data;
- storing message indices to enable the subtransaction data to be more efficiently recovered.

In the end, `libclio` (and parts of `libxp`) provided approximately 88% of the total lines of code in `mnemd`.

Chapter 6

Conclusions

```
cd $FORUM
find include code (-name "*.c" -o -name "*.h") -print > files
c_count -l -s ‘cat files‘
Grand total
----------------
  7113  lines had comments        26.3 %
   388  comments are inline       -1.4 %
   947  lines were blank           3.5 %
  1538  lines for preprocessor     5.7 %
 17825  lines containing code     65.9 %
 27035  total lines              100.0 %
```

In some real sense, the above lines of code (LOC) statistics for the implementation of Forum are the main result of this dissertation, inasmuch as they demonstrate the following:

feasibility

Implicitly the LOC counts show that the implementation can be accomplished, albeit with a significant amount of work.

affordability

Explicitly, the LOC counts argue that the implementation can be done economically. The productivity of programmers varies, depending on the skill level of the programmer, the complexity of the task, the required quality level of the code, and the suitability and sophistication of the programming tools. Estimates usually range between five and twenty-five LOCs per day.

Since most of the Forum components involve fairly sophisticated programming (large integer mathematics, cryptographic algorithms, transaction management and a non-trivial network protocol), a figure from the lower end of this range, 10 LOCs per

day, is probably appropriate. And an effective work year of 220 days also seems reasonable, after vacations, holidays, business trips and other occupational issues.

With these assumptions, this amount of code would take around eight programmer-years.[1] In the USA at present, the care, feeding and maintenance of a skilled programmer costs between $150K and $200K. Even at the higher figure, the implementation of Forum would cost around $1.6M. And if off-the-shelf libraries were used for the mathematics and cryptography components, the cost would be reduced since the clio specific code amounted to about 8000 lines of code.

proportionality

The LOCs also demonstrate that the size of the implementation is not out of proportion to other components in the system. The LOC statistics for the Linux 2.4.29 kernel are:

```
Grand total
----------------
 15265  lines had comments        23.6 %
  1737  comments are inline       -2.7 %
  8975  lines were blank          13.9 %
  5068  lines for preprocessor     7.8 %
 37022  lines containing code     57.3 %
 64593  total lines              100.0 %
```

But this does not include device drivers (1,649,387 LOCs)[2] or networking (186,373 LOCs) or any modules. And to compare a similar type of component, the LOC statistics for the SSL library in Linux are:

[1] It took me about five years, including thorough testing of the mathematics and cryptographic libraries.
[2] The IEEE1394 device driver alone is 18,339 LOCs.

```
Grand total
----------------
  6414  lines had comments        20.3 %
   564  comments are inline       -1.8 %
  3146  lines were blank          10.0 %
  2646  lines for preprocessor     8.4 %
 19952  lines containing code     63.2 %
 31594  total lines              100.0 %
```

Note that this does not include the code for any of the cryptographic algorithms, and while SSL contains a complete security protocol, it does not include specific electronic commerce functionality.

What the implementation cannot demonstrate is the **utility** of Forum. This could be argued empirically by implementing a significant subset of electronic commerce applications on Forum. But absent some authoritative way to determine which applications are significant, this approach is open to charges of speciousness.

A better argument for utility is the support in Forum for the structural properties of quantitative and qualitative scalability described in chapter 2. These properties enable each Forum transaction to be able to have an arbitrary number of participants, each with an unrestricted role.

These properties are supported in Forum by allowing each transaction to be divided into separate, equipollent two-party subtransactions, and by not constraining the number or content of messages within each subtransaction. The current implementation limits the number of subtransactions to 65,535 (or 131,070 or so participants), and each subtransaction can include over two billion messages, each with a length of over two billion bytes.

The transaction protocol in Forum, clio, is more comparable to TCP or UDP than it is to protocols like SET [21] or NetBill [33]. These protocols support specific subsets of transactions, for the most part, rather than general transaction properties. Just as TCP

and UDP are designed to support protocols with specific functionality like HTTP, FTP or SNMP, clio is designed to complement purely functional protocols like EDI [32].

What, if anything, the future holds for Forum is difficult to say with any certainty. The main things missing from the implementation of Forum are the key server, hephaistos, and the use of a proper name server. The current implementation of the transaction server is also quite limited in that it does not preserve transaction state across server restarts. A simple scheme for presistent transaction state would use the local filesystem, a better scheme would use a database manager, and a more interesting scheme would use the durability server, perhaps in a separate subtransaction within the transaction whose state is being preserved. Although the current implementation does not achieve the universality of the original Roman Forum, that, too, wasn't built in a day.

Finally, note that while Forum includes support for the contractual properties of mutual obligation, agreement, consideration and durability, the successful conclusion of a Forum transaction does not, in and of itself, constitute a contract. Maine wrote that in the earliest stages in the development of the law,

> *. . . substantive law has at first the look of being gradually secreted in the interstices of procedure;* [22, p 215]

The main intent of Forum, at this early point in the development of electronic commerce, is to make those interstices smaller.

Bibliography

[1] J.T. Abhy and Bryan Walker, editors. *The Commentaries of Gaius*. The University Press, Cambridge, 1870.

[2] J.T. Abhy and Bryan Walker, editors. *The Institutes of Justinian*. The University Press, Cambridge, 1876.

[3] P. Baldy, H. Dicky, R. Medina, M. Morvan, and J.F. Vilarem. Efficient Reconstruction of the Causal Relationship in Distribued Systems. Technical Report 92-013, Laboratorie d'Informatique, de Robotique et de Microelectronique de Montpellier, France, June 1992.

[4] Phillip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, Reading, MA, 1987.

[5] Kenneth Birman and Keith Marzullo. The Role of Order in Distributed Programs. Technical Report TR 89-1001, Cornell University, Ithaca, NY, 1989.

[6] Kenneth P. Birman and Thomas A. Joseph. Reliable Communications in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[7] W. H. Buckler. *The Origin and History of Contract in Roman Law*. C. J. Clay & Sons, Ave Maria Lane, London, 1895.

[8] B. Charron-Bost. Concerning the Size of Logical Clocks in Distributed Systems. *Information Processing Letters*, (39):11–16, July 1991.

[9] Dorothy Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, MA, 1983.

[10] C. J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Order. In *Proceedings 11th Australian Computer Science Conference*, pages 55–66. University of Queensland, February 1988.

[11] Jerry Fowler and Willy Zwaenepoel. Causal Distributed Breakpoints. In *Proceedings 10th International Conference on Distributed COmputing Systems*, pages 134–141, May 1990.

[12] Mohamed Gouda. *Elements of Network Protocol Design*. John Wiley & Sons, Inc., New York, NY, 1998.

[13] Michael Grant. *The Roman Forum*. The Macmillan Company, New York, 1970.

[14] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1993.

[15] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, London, 1984.

[16] H. F. Jolowicz. *An Historical Introduction to Roman Law*. The University Press, Cambridge, 1932.

[17] David Kahn. *The Codebreakers*. Scribner, New York, NY, 1996.

[18] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security: Private Communication in a Public World*. Prentice Hall, Englewood Cliffs, NJ, 1995.

[19] Henry F. Korth and Abraham Silbershatz. *Database Systems Concepts*. McGraw Hill, New York, 3rd edition, 1997.

[20] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[21] Larry Loeb. *Secure Electronic Transactions*. Artech House, Norwood, MA, 1998.

[22] Henry Sumner Maine. *Ancient Law*. Charles Scribner & Co., No. 654 Broadway, New York, First American – from Second London edition, 1871.

[23] Henry Sumner Maine. *Early Law and Custom*. John Murray, Albermarle Street, London, 1907.

[24] William Morris, editor. *The American Heritage Dictionary of the English Language*. Houghton Mifflin Company, Boston, 1969.

[25] Barry Nicholas. *An Introduction to Roman Law*. Oxford University Press, Oxford, 1962.

[26] Larry L. Peterson, Nick C. Buchholz, and Richard D. Schlichting. Preserving and Using Context Information in Interprocess Communication. Technical Report TR 88-23, The University of Arizona, Tucson, AZ, 1988.

[27] Roscoe Pound. *Introduction to the Philosophy of Law*. Yale University Press, New Haven, 1922.

[28] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, March 1975.

[29] Bruce Schneier. *Applied Cryptography*. John Wiley and Sons, Inc., New York, NY, second edition, 1996.

[30] Reinhard Schwarz and Friedemann Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. Technical Report SFB 124-15/92, University of Kaiserlautern, Kaiserlautern, Germany, 1992.

[31] S. P. Scott, editor. *The Civil Law*, volume 1-17. The Central Trust Company, Cincinnati, 1932.

[32] Mostafa Hashem Sherif. *Protocols for Secure Electronic Commerce*. CRC Press, New York, 2000.

[33] M. Sirbu and J. D. Tygar. Netbill: An Internet Commerce System Optimized for Network Delivered Services. *IEEE Personal Communications*, 2(4):34–39, August 1995.

[34] Sean W. Smith and J. D. Tygar. Signed Vector Timestamps: A Secure Protocol for Partial Order Time. Technical Report CMU-CS-93-116, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, February 1993.

[35] M. Spezialetti and J.P. Kearns. An Efficient Implementation of Vector Clocks. *Information Processing Letters*, (43):47–52, August 1992.

[36] William Stallings. *Cryptography and Network Security*. Prentice Hall, Upper Saddle River, NJ, second edition, 1998.

[37] Douglas H. Steves. Specifying Security Policy in Computer Systems. See the author.

[38] Douglas H. Steves, Chris Edmondson-Yurkanan, and Mohamed Gouda. A Protocol for Secure Transactions. In *The Second USENIX Workshop on Electronic Commerce*, pages 201–212, November 1996.

[39] Douglas H. Steves, Chris Edmondson-Yurkanan, and Mohamed Gouda. Properties of Secure Transaction Protocols. *Computer Networks and ISDN Systems*, 29(4):1809–1821, November 1997.

[40] Douglas H. Steves, Chris Edmondson-Yurkanan, and Mohamed Gouda. An ACID Framework for Electronic Commerce. In *The First International Conference on Telecommunications and Electronic Commerce*, pages 51–65, November 1998.

[41] Homer Thompson and R. E. Wycherley. *The Agora of Athens*. The American School of Classical Studies at Athens, Princeton, N.J., 1972.

[42] Russ VerSteeg. *Law in the Ancient World*. Carolina Academic Press, Durham, N.C., 2002.

[43] Alan Watson. *The Law of Obligations in the Later Roman Republic*. Oxford University Press, Oxford, 1965.

[44] Alan Watson. *Rome of the XII Tables*. Princeton University Press, Princeton, 1975.

[45] Alan Watson, editor. *The Digest of Justinian*. University of Pennsylvania Press, Philadelphia, 1985.

[46] Raymond Westbrook, editor. *A History of Ancient Near Eastern Law*, volume 1-2. Brill, Boston, 2003.

[47] Thomas Woo and Simon Lam. Authorization in Distributed Systems: A Formal Approach. In *Proceedings of the 13th IEEE Symposium on Research in Security and Privacy*, pages 33–50. IEEE, May 1992.

Vita

Douglas Howard Steves was born in Erie, Pennsylvania, and graduated from Whitmer Senior High School in Toledo, Ohio, in 1974. He received a B.A. degree in Classics and a B.S. degree in Mathematics from the University of Toledo in 1984. He received a M.A. degree in Computer Sciences from the University of Texas at Austin in 1992.

From 1984 to 1993, he was employed by IBM working in system research and design. From 1993 to 1994 and from 1997 to 2000, he was employed by Renaissance Softworks as a consultant and programmer. From 1994 to 1995, he was employed by Isis Distributed Systems as a consultant. From 1995 to 1997, he was employed by Skipstone working in system design and implementation.

Permanent Address: P.O.Box 201743

Austin, TX 78720-1743

This dissertation was typeset with LaTeX 2ε[3] by the author.

[3] LaTeX 2ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.