Constructing Atomic Commit Protocols Using Knowledge

Weihai Yu

Department of Computer Science
University of Tromsø, Norway
weihai@cs.uit.no

Abstract

The main purpose of this paper is to use logic of process knowledge as a tool to (1) obtain better understanding of the atomic commit problem, to (2) analyze at a rather intuitive level the rationale behind correct atomic commit protocols that are in use today, and to (3) conduct the design and construction of atomic commit protocols. This is not a theoretic paper, although some formal definitions are introduced and used in the discussions. This paper is intended for people not familiar with logic of process knowledge, or even atomic commit protocols.

1 Introduction

Atomic commit protocols are a key element in supporting global atomicity of distributed transactions. *Two-phase commit protocol* (*2PC*) is the de facto standard atomic commit protocol [1][5]. As web services are gaining importance in open distributed processing, 2PC becomes a hot research topic again. It is widely agreed that 2PC is important to guarantee correctness properties in the complex distributed world whilst at the same time it reduces parallelism due to high disk and message overhead and locking during windows of vulnerability. There are a number of optimizations to the basic 2PC (e.g., [2][4][9][11][15]). Some of them are so widely used that they are built into commercial systems and become part of the standards for distributed transaction processing [14][16].

To understand and manipulate the vast amount of atomic commit protocols (including various 2PC variations) is not an easy task. This paper is an attempt to bring a useful picture of this important research area with the help of the logic of process knowledge.

2 A General Model of Distributed Systems

This model is typically used in the study of knowledge and coordination [3][6][10][13].

A distributed system is a finite set *P* of *n* processes connected by a communication network. We assume the existence of a *global time* of natural numbers, which is only for the purpose of the description of system executions and thus is not observable by the processes. At each moment in time, a number of processes may each execute an *event*, which can be a local event, sending or receiving a message, and so on. Local events may include crash failure and restart thereafter, timeout, and local actions specific to the system, as we shall see later for atomic commit protocols.

The *local state* of a process consists of its initial state at time 0 and the sequence of events it has executed. The *global state* of a system is an n+1 tuple (s_e, s_1, …, s_n) where s_e is the *operating* *environment* of the system and $s_1, \dots, s_n$ are local states of processes. The operating environment is used to encapsulate the characteristics of the system that are not captured in the local states, such as the failure model of the communication network. Like the global time, the operating environment is only for the purpose of description of system behavior and is not observable by the processes. For example, a network failure can be modeled as a partition event of the operating environment.

A *protocol* is a function from local states to actions (i.e., local actions and message sendings). Note that processes do not run their protocols in isolation. It is the combination of the protocols run by all processes that causes the system to behave in a particular way. A *joint protocol PT* is a tuple $(PT_1, \dots, PT_n)$ consisting of protocols PT_i for each of the processes $i = 1, \dots, n$.

Possible joint behavior of the system over time is modeled by *runs*. A run is a function from time to global states. Thus in a possible execution r, $r(0)$ is the initial global state of the system, $r(t)$ is the global state at time t, and so on. The behavior of the system running joint protocol PT is characterized by the set of all possible runs, denoted by R_{PT}, or R if PT is obvious from context. If run $r \in R$ and t is a natural number, $r(t)$ is a *point* in R.

If $r(t) = (s_e, s_1, \dots, s_n)$, we denote $r_e(t) = s_e$ and $r_i(t) = s_i$ for $i = 1, \dots, n$. Two points $r(t)$ and $r'(t')$ are *indistinguishable* to process i, denoted $r(t) \sim_i r'(t')$, if $r_i(t) = r'_i(t')$, i.e., if the process i has the same local state at both points. We write a -| r if event a occurs in run r. Similarly, a-| $r(t)$ or a -| $r_i(t)$ if a occurs in a point of a run or in a local state of process i.

Next, we describe some events of the distributed system that will be used in the discussions later. Let P be the set of processes and $p, q \in P$, we have the following events:

- $\text{send}_p(q, m)$, p sends a message m to q,
- $\text{receive}_p(q, m)$, p receives a meaage m from q,
- fail_p, p fails,
- restart_p, p restarts after a failure,
- $\text{timeout}_p(a)$, p timeouts after an event a,
- partition_e (U), a network (re-)partition occurs. U is a partition of the processes. |U| is the number of islands of the partition. |U| = 1 if there is no network partition. A message from p to q can only be delivered when p and q are in the same island.

We can define well-formedness of runs, for example the following (we will not give a complete definition here):

- The only possible event after fail_p is restart_p,
- For $t' > t$, $\text{send}_p(q, m)$ -| $r(t) \wedge \text{receive}_q(p, m)$ -| $r(t') \Rightarrow p$ and q are in the same partition island between t and t'.

3 Process Knowledge

Process knowledge was first defined by Halpern and Moses [7] and detailed in [3].

To describe the semantics of a protocol, we assume a set Φ of primitive propositions on points of runs, describing basic facts about the system. A fact φ is either true or false at a given point $r(t)$ in R, denoted $(R, r, t) \models \varphi$ and $(R, r, t) \not\models \varphi$, respectively. The set of well-formed formulae WFF is such that $\Phi \subseteq$ WFF, and for any φ and $\varphi' \in$ WFF, $p \in P$ set of processes and a group $G \subseteq P$, $\neg\varphi$, $\varphi\wedge\varphi'$, $\varphi\vee\varphi'$, $\Diamond\varphi$ (eventually φ), $K_p\varphi$ (p knows φ), $E_G\varphi$ (everybody in group G knows),

$S_G\varphi$ (somebody in G knows), $D_G\varphi$ (it is distributed knowledge in G), $C_G\varphi$ (it is common knowledge in G) are also in WFF. We often omit the subscript G if $G = P$.

Given the events defined earlier, we can define the propositions corresponding to the occurrence of these events. For example:

- $FAIL_p$, the last event of p is $fail_p$,
- FAIL, either $FAIL_p$ for some p or partition $|U| > 1$.
- NO_FAIL = $\neg$FAIL.
- NO_FAIL (t, t'), no fail between time t and t'.

Below are the formal definitions of some knowledge operators:

- $(R, r, t) \models \Diamond\varphi$ iff $(R, r, t') \models \varphi$ for some $t' \geq t$.
- $(R, r, t) \models K_p\varphi$ iff $(R, r', t') \models \varphi$ for all r' (t') such that $r(t) \sim_p r'(t')$.
- $(R, r, t) \models E_G\varphi$ iff $(R, r, t) \models K_q\varphi$ for all $q \in G$.
- $(R, r, t) \models S_G\varphi$ iff $(R, r, t) \models K_q\varphi$ for some $q \in G$.
- $(R, r, t) \models D_G\varphi$ iff $(R, r', t') \models \varphi$ for all r' (t') such that $r(t) \sim_G r'(t')$.
- $(R, r, t) \models C_G\varphi$ iff $(R, r, t) \models E_G\varphi$, and $(R, r, t) \models E_GE_G\varphi$, and …

Another way to define common knowledge is that it is the greatest fixed point of $X = E_G(\varphi \wedge X)$.

In many situations, common knowledge is too strong a notion to be achievable. In some cases, the weaker notion of eventual common knowledge is used instead. Eventual common knowledge, denoted $C^{\Diamond}\varphi$, is the greatest fixed point of $X = \Diamond E_G(\varphi \wedge X)$.

Furthermore, a proposition is said to be *stable* if once true, it remains true forever. Formally, if $(R, r, t) \models \varphi$, then for all $t' \geq t$, $(R, r, t') \models \varphi$

4 The Atomic Commitment Problem

Informally, an atomic commitment problem requires processes to agree on a common outcome which can be either *Commit* or *Abort*. More specifically, an atomic commit protocol must guarantee the *atomic commitment properties* [1]:

- AC1: All processes that reach an outcome reach the same one.
- AC2: A process cannot reverse its outcome after it has reached one.
- AC3: The *Commit* outcome can only be reached if all participants voted *Yes*.
- AC4: If there are no failures and all participants voted *Yes*, then the outcome will be *Commit*.
- AC5: Consider any execution containing only failures that the protocol is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach an outcome.

Next, we specify more formally the atomic commit problem using the logic of process knowledge. We first define some local actions specific to atomic commit protocols, for $p \in T$ participant processes in the transaction:

- yes_p, p votes *Yes*,

- no_p, *p* votes *No*,
- $commit_p$, *p* commits,
- $abort_p$, *p* aborts.

The corresponding primitive propositions after the executions of the actions are: YES_p, NO_p, $COMMIT_p$, $ABORT_p$.

Formally, the atomic commitment properties are:

- AC1: $\forall p, q \in T, (R, r, t) \models \neg(COMMIT_p \wedge ABORT_q)$
- AC2: implied by AC1 and that $COMMIT_p$ and $ABORT_p$ are stable.
- AC3: $(R, r, t) \models COMMIT_p \Rightarrow (R, r, t) \models \wedge_{q\in T} YES_q$
- AC4: $NO\_FAIL \wedge (R, r, t) \models \wedge_{p\in T} YES_p \Rightarrow (R, r, t) \models \Diamond(\wedge_{p\in T} COMMIT_p)$
- AC5: For a *d* sufficiently large, if NO_FAIL(*t*, *t*+*d*), then there is *t'* between *t* and *t*+*d*, $(R, r, t') \models \wedge_{p\in T} COMMIT_p \vee \wedge_{p\in T} ABORT_p$

5 Constructing Atomic Commit Protocols Using Knowledge

We now discuss how process logic could be used to the specification and construction of atomic commit protocols. We first look at the atomic agreement requirement, which is the main focus of many papers in the literature [6][8][12], and then the protocol termination requirement, which is also very important [13] but not much research result is available. Finally we discuss the construction of the protocols.

5.1 Agreement

Neiger [13] showed a necessary condition for all participants to achieve the *Commit* agreement:

$$(R, r, t) \models COMMIT_p \Rightarrow (R, r, t) \models K_p(\wedge_{q\in T} YES_q)$$

That is, a process commits only if it knows that all participants have voted *Yes*. This is obvious and several papers used this to show the flexibility of using knowledge to achieve agreement [6][8][12].

However, this necessary condition is not sufficient to achieve the *Commit* agreement. For example, in a naïve decentralized two-phase commit protocol, every participant broadcasts a Y*es* vote to all other participants. If the network is not reliable, some of the participants may receive all *Yes* votes but some may not. Although it may not necessarily lead to inconsistent outcome if those participants that do receive all *Yes* votes decide to commit, obviously something in addition must be done for those participants that do not receive all *Yes* votes.

AC4 presented a sufficient condition for all participants to achieve the *Commit* agreement. For every individual participant, it suffices to decide the *Commit* outcome when it knows that all participants voted *Yes* and no failure has occurred:

$$NO\_FAIL \wedge (R, r, t) \models K_p(\wedge_{q\in T} YES_q) \Rightarrow (R, r, t) \models COMMIT_p$$

This sufficient condition could be useful in special circumstances. For example, given atomic broadcast, *p* knowing YES_q (where $p \neq q$) implies NO_FAIL:

$$(R, r, t) \models K_p YES_q \Rightarrow NO\_FAIL$$

Thus it is both necessary and sufficient to decide *Commit* when a process receives *Yes* votes from all participants:

$$(R, r, t) \models \mathrm{COMMIT}_p \Leftrightarrow (R, r, t) \models K_p(\wedge_{q\in T}\mathrm{YES}_q)$$

In general, however, there is a gap between the necessary and the sufficient conditions for the *Commit* agreement. If there is a condition that is both necessary and sufficient for the individual processes to decide the *Commit* outcome, it should look like this:

$$(R, r, t) \models K_p(\wedge_{q\in T}\mathrm{YES}_q \wedge \mathrm{LESS\_FAIL}) \Leftrightarrow (R, r, t) \models \mathrm{COMMIT}_p$$

And for both *Commit* and *Abort* outcomes, a more general specification should look like:

- $(R, r, t) \models \mathrm{COMMIT}_p \Leftrightarrow (R, r, t) \models K_p(\wedge_{q\in T}\mathrm{YES}_q \wedge \mathrm{LESS\_FAIL})$
- $(R, r, t) \models \mathrm{ABORT}_p \Leftrightarrow (R, r, t) \models K_p(\vee_{q\in T}\mathrm{NO}_q \vee \mathrm{MORE\_FAIL})$

There are some difficulties, though, to apply this specification in practice. For example, if LESS_FAIL and MORE_FAIL overlap, the specification is non-deterministic and the protocol would not guarantee atomicity. More importantly, LESS_FAIL is hard to define in theory and detect at the processes in practice.

In Neiger [13], conditions like $(\wedge_{q\in T}\mathrm{Yes}_q \wedge \mathrm{LESS\_FAIL})$ and $(\vee_{q\in T}\mathrm{No}_q \vee \mathrm{MORE\_FAIL})$ are called *enabling conditions* for *Commit* and *Abort* decisions. However, how the enabling conditions should be assigned to practical atomic commit protocols is not discussed.

In what follows, we name the enabling conditions COMMIT_ENABLED and ABORT_ENABLED respectively. That is,

- $\mathrm{COMMIT\_ENABLED} \equiv \wedge_{q\in T}\mathrm{YES}_q \wedge \mathrm{LESS\_FAIL}$
- $\mathrm{ABORT\_ENABLED} \equiv \vee_{q\in T}\mathrm{NO}_q \vee \mathrm{MORE\_FAIL}$

Practically in most popular practical atomic commit protocols, COMMIT_ENABLED is implicitly defined as "somebody knows the fact" (S_T COMMIT_ENABLED) and typically (K_c COMMIT_ENABLED) where this "somebody" is a particular *c*, known as the *coordinator* of the protocol. That is, the decision made by the coordinator is tantamount to coordinator's knowledge of the enabling conditions:

- $(R, r, t) \models \mathrm{COMMIT}_c \Leftrightarrow (R, r, t) \models K_c\ \mathrm{COMMIT\_ENABLED}$
- $(R, r, t) \models \mathrm{ABORT}_c \Leftrightarrow (R, r, t) \models K_c\ \mathrm{ABORT\_ENABLED}$

In other words, LESS_FAIL is defined by the fact that the coordinator has successfully received all *Yes* votes within a specific deadline. Failures after that are considered to be LESS_FAIL. All other failures are considered to be MORE_FAIL, including any participant failing to give a vote or any vote failing to arrive at the coordinator in time.

More formally,

- $(R, r, t) \models K_c\ \mathrm{COMMIT\_ENABLED} \equiv (R, r, t) \models K_c(\wedge_{q\in T}\mathrm{YES}_q \wedge \neg\mathrm{timeout}_c(\mathrm{b}))$
- $(R, r, t) \models K_c\ \mathrm{ABORT\_ENABLED} \equiv$

$$(R, r, t) \models K_c(\vee_{q\in T}\mathrm{NO}_q \vee (\neg\wedge_{q\in T}\mathrm{YES}_q \wedge \mathrm{timeout}_c(\mathrm{b})\ )$$

where b is an earlier event local at *c*, such the beginning of the protocol.

Given the role of the coordinator as described above, the next task of the protocol is to ensure that all participants $p \in T$ eventually knows the enabling condition and reach the corresponding outcome:

$(R, r, t) \models \mathrm{COMMIT}_c \Rightarrow (R, r, t) \models \Diamond K_p\ \mathrm{COMMIT}_c$

$\Leftrightarrow (R, r, t) \models \Diamond K_p\ \mathrm{COMMIT\_ENABLED} \Leftrightarrow (R, r, t) \models \Diamond \mathrm{COMMIT}_p$

5.2 Termination

An atomic commit protocol terminates when all participants reach the same outcome:

$(R, r, t) \models \mathrm{COMMIT\_END} \vee \mathrm{ABORT\_END}$

where $\mathrm{COMMIT\_END} \equiv \wedge_{p \in T}\mathrm{COMMIT}_p$ and $\mathrm{ABORT\_END} \equiv \wedge_{p \in T}\mathrm{ABORT}_p$

Note that this is distributed knowledge among the participants, or formally

$(R, r, t) \models D_T\ (\mathrm{COMMIT\_END} \vee \mathrm{ABORT\_END})$.

For every individual participant to determine the coordinated outcome, Neiger [13] has shown the necessary condition for termination:

- $(R, r, t) \models \mathrm{COMMIT}_p \Rightarrow (R, r, t) \models K_p C^{\Diamond}\ \mathrm{COMMIT\_ENABLED}$
- $(R, r, t) \models \mathrm{ABORT}_p \Rightarrow (R, r, t) \models K_p\ C^{\Diamond}\ \mathrm{ABORT\_ENABLED}$

That is, every individual participant must know that the enabling condition is eventual common knowledge among the participants.

Similar to the necessary condition for agreement, this could be useful in special circumstances when NO_FAIL is used instead of LESS_FAIL in COMMIT_ENABLED. For example, given atomic broadcast,

$(R, r, t) \models K_p \mathrm{YES}_q \Rightarrow \mathrm{NO\_FAIL}$

Therefore

$(R, r, t) \models C^{\Diamond}(\wedge_{q \in T}\mathrm{Yes}_q) \Rightarrow (R, r, t) \models C^{\Diamond}\ \mathrm{COMMIT\_ENABLED}$

To obtain eventual common knowledge is a difficult task in general (though easier than obtaining common knowedge). Here again, the use of a coordinator could be a rescue. Assume that it is *a priori* common knowledge among participants of a transaction that a coordinator can guarantee the following for a proposition φ:

$(R, r, t) \models K_c\varphi \Rightarrow (R, r, t) \models \Diamond C^{\Diamond}\varphi$

That is, the coordinator's knowledge about φ will eventually lead to eventual common knowledge among the participants. If we replace φ with the enabling condition of an outcome, then, in order to achieve $K_p C^{\Diamond}\varphi$, it suffices to obtain $K_p K_c \varphi$. This coincides with the sufficient condition for achieving agreement among the participants. So the coordinator plays double roles: (1) to detect the enabling condition for *Commit*, and (2) to ensure the termination of the protocol.

Of course, it remains as the next task for the coordinator and the participants to guarantee that this assumption actually holds.

5.3 Designing atomic commit protocols

We start with a design strategy which leads to protocol structures. Every protocol structure then allows for multiple implementations.

According to the discussions in the previous sections, a useful strategy to design an atomic commit protocol is to use a coordinator and divide the protocol into multiple steps, such as the following:

1. $(R, r, t_1) \models K_c\varphi$
2. $(R, r, t_2) \models K_pK_c\varphi$
3. $(R, r, t_3) \models K_cC^{\Diamond}\varphi$

where *c* is the coordinator, φ is the enabling condition of either *Commit* or *Abort*, and $t_1 \leq t_2 \leq t_3$.

The first two steps are actually the two phases in the well-known *Two-Phase Commit* protocols (2PC) that are used to achieve the agreement: the first vote-collection phase and the second outcome-notification phase. In the last step, the coordinator ensures that the protocol will eventually terminate. Practically, this means that the coordinator must keep the necessary information stably until this goal is achieved.

One practical design for the last termination step is that the coordinator keeps the necessary information until it knows that the protocol actually has terminated. That is:

$$(R, r, t_3) \models K_c(\mathrm{COMMIT\_END} \vee \mathrm{ABORT\_END})$$

This is known as the *baseline 2PC*, or *presumed nothing 2PC.*

Due to the fact that there can only be one of the two outcomes *Commit* and *Abort*, an alternative design is that the coordinator keeps the necessary information until the following:

- $(R, r, t_3) \models K_c(\mathrm{COMMIT\_END} \vee \Diamond\mathrm{ABORT\_END})$
- $(R, r, t_3) \models K_c(\Diamond\mathrm{COMMIT\_END} \vee \mathrm{ABORT\_END})$

That is, the coordinator keeps the necessary information about the transaction until it knows that either the protocol has actually committed (aborted) or eventually all participants will only abort (commit). These are known as the *presumed abort* (*presumed commit*) *2PC.* The presumption is the knowledge *a priori* known by the system. The advantage of the presumed outcome 2PC is that the coordinator does not have to collect all the information about the actual termination of the protocol in both *Commit* and *Abort* cases. For the presumed outcome, it is safe for the coordinator to be information-free about the transaction and still be able to guarantee the correct termination of the protocol. Of course, care must be taken to ensure that when the coordinator is information-free about a transaction, it does know that, either has the transaction terminated with the non-presumed outcome, or the only possible outcome will be the presumed one.

In general, the design of a distributed protocol can be done by first specifying the states of knowledge and then providing the implementation of the state transitions. This is similar to the use of formal specifications like VDM and Z in the design of sequential software. Janssen [8] gave a classification of some common abstract transitions among states of knowledge and discussed how they can be realized in different contexts.

For the baseline presumed nothing 2PC sketched above, we can present the corresponding states of knowledge as bellow:

$$\mathrm{VOTE}_p = \mathrm{YES}_p \vee \mathrm{NO}_p$$

$$\mathrm{DECISION}_c = \mathrm{COMMIT}_c \vee \mathrm{ABORT}_c$$

$$\mathrm{COMMIT}_c = K_c(\wedge_{p \in T}\,\mathrm{YES}_p)$$

$$\mathrm{ABORT}_c = K_c(\vee_{p \in T}\,\mathrm{NO}_p \vee (\mathrm{TIMEOUT}_c(\text{some local event at c}))$$

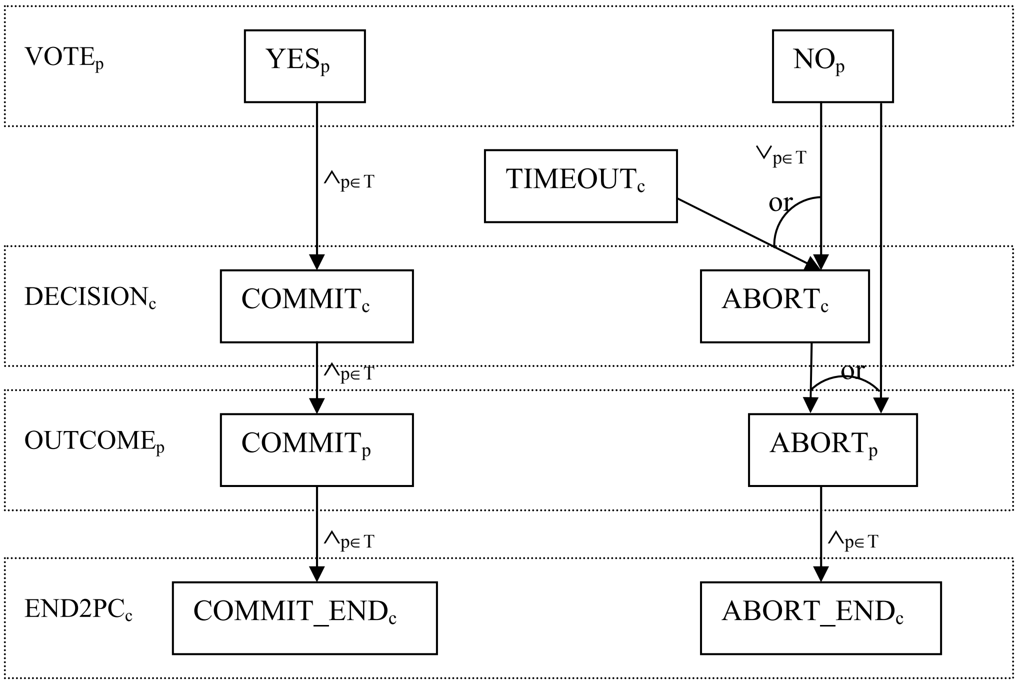

Figure 1. Knowledge dependency graph of presumed nothing 2PC

$OUTCOMT_p = COMMIT_p \vee ABORT_p$

$COMMIT_p = K_pCOMMIT_c$

$ABORT_p = K_p(ABORT_c \vee NO_p)$

$END2PC_c = COMMIT\_END_c \vee ABORT\_END_c$

$COMMIT\_END_c = K_c (\wedge_{p\in T} COMMIT_p)$

$ABORT\_END_c = K_c (\wedge_{p\in T} ABORT_p)$

Basically, the states of knowledge and transitions among them form a *knowledge-dependency graph*. For example, the state transitions for the baseline presumed nothing 2PC corresponds to the graph in Figure 1. This knowledge dependency graph provides a basic structure of a correct (i.e., meets the agreement and termination requirements) atomic commit protocol.

5.4 Implementing atomic commit protocols

Decoupling specification from implementation enables flexibility. Here we discuss some of the implementation options.

5.4.1 Stable knowledge and recovery

One important goal in protocol design is to optimize performance and minimize runtime overhead. In atomic commit protocols, network messages and disk accesses are the most significant runtime overhead.

In our discussions so far, we require that the propositions (or knowledge states) are stable, i.e., once a fact becomes true, it will remain true forever. However in our system model, processes are subject to failure at any possible moment and can restart thereafter. This implies that every fact must be stably logged, which is a tremendous runtime overhead.

In practice however, facts need not be stable forever. Data about a fact that is not needed any more, can be garbage collected. In fact, one important requirement is that all data about a transaction *must* be eventually garbage collected.

Basically, the lifetime of data about a fact is bounded to its usage in knowledge ascription. For example, YES_p will contribute to $COMMIT_c$ and $\neg NO_p$, so its lifetime is bounded to the lifetime of NO_p and the time at which $COMMIT_c$ is known. Note that the lifetime of mutually exclusive facts are always bounded together, such as YES_p and NO_p. So it suffices to only consider the lifetime of $VOTE_p$.

Some facts need not be logged at all. For example, the fact $FAIL_p$ will persist as long as the process remains failed. As soon as the process restarts and the necessary information is re-established, the fact that it has failed may not be of interest any more.

Some facts are subsumed by other facts. For example, it is always safe to assume that the time between the events $fail_c$ and $restart_c$ is greater than the value set in event $timeout_c$, so $TIMEOUT_c$ is implied by $FAIL_c$ and need not be logged.

Some local facts hang closely with one another and can be stably recorded in a single log. For example, NO_p directly leads to $ABORT_p$, so the two facts can be recorded in a single log.

5.4.2 Knowledge and messages

In the theory of process knowledge [10], it is known that if at time t_1 a fact φ local to process q is unknown to process p and at time $t_2 > t_1$ p knows φ, then there is a message chain from q to p between t_1 and t_2. This means that in the knowledge dependency graph, if there is an edge between knowledge of different processes, then in protocol executions corresponding to the graph, there is a message chain between the two processes. How this message chain occurs is an implementation issue. Here we have at least the following choices:

- Pull and push of knowledge.

  This applies to virtually every piece of knowledge in the graph. For votes, typically a pull model is applied, where the coordinator pulls this knowledge from the participants with a *Prepare* message. In certain optimizations, a push model is used instead, such as the 2PC with *Early Prepare* and *Unsolicited Yes-Vote*s. Decisions, on the other hand, are typically pushed from the coordinator to the participants. However, if a participant does not hear from the coordinator for too long, it may pull this knowledge from the coordinator by sending an inquiry message.

- Knowledge collection and notification through a particular communication topology.
  - Stars. All messages go through the coordinator. This is quite natural in our example, because all communication is in fact between the coordinator and the participants. This version of 2PC with direct communication between the coordinator and the participants is known as the *flat 2PC*.
  - Hierarchy. Processes are organized in a tree, typically rooted by the coordinator. This is also known as *tree 2PC* or *2PC with interposition*, which is most widely used. One important reason for its wide use is that this tree structure usually overlaps with the remote invocation structure of the application.
  - Linear. This can be considered a special form of hierarchy.

    - Ring. This is not very much explored in practice, basically because establishing such a ring structure per transaction is expensive. However, if there are a lot of transactions with the same set of participants, this ring structure can be established once and used many times. In such circumstances (which is often the case in many applications), advantages of ring-structured communication can be obtained. Advantages include: smaller number of messages, predictable upper bound of transfer time of messages through all participants, piggyback of several messages into one token through the same structure, etc.

The message chain theory has some other important implications to the implementation of an atomic commit protocol. Here we discuss two of them.

In the knowledge dependency graph, a knowledge state may depend on some other knowledge states. In other words, there can be multiple paths toward the same knowledge state. The number of paths, however, may change during the execution of the protocol. For example, there are two possible paths leading to END2PC_c at the beginning of the protocol: COMMIT_p or ABORT_p, both from all participants. When the fact COMMIT_c is reached, the number of paths reduces to only COMMIT_p from all participants. For OUTCOME_p, at the beginning, it depends on COMMIT_c, ABORT_c or NO_p. After voting *Yes*, the number of paths reduces to COMMIT_c and ABORT_c. We can describe such dependency with a *dependency set* of a knowledge state, which is a set of multiple subsets of processes. For END2PC_c, it is $\{T\}$, both at the beginning and after COMMIT_c is reached. For OUTCOME_p, it is $\{\{c\}, \{p\}\}$ at the beginning and $\{\{c\}\}$ after a *Yes* vote. We say that END2PC_c is *blocked* upon T, because the knowledge END2PC_c will not be obtained until all processes in T are available (not necessary at the same time). Note that OUTCOME_p's dependency set is $\{\{c\}, \{p\}\}$ at the beginning. This means that OUTCOME_p can be obtained based on some local knowledge (here NO_p) only. So OUTCOME_p is not blocking at the beginning. It is, however, blocked upon $\{c\}$ after having voted *Yes*.

Blocking is an undesirable property of the protocol. It is more serious for some knowledge state than other and deserves more concern. In 2PC, the blocking effect on OUTCOME_p (after *Yes* vote) is more serious than the blocking effect on END2PC_c. The main reason is that a participant cannot release locks on data resources until it knows the outcome of a transaction. In fact, 2PC is known as a blocking protocol due to this particular blocking effect. In general, the blocking effect can be ameliorated in two ways:

- Enhanced implementation, by improving the availability of the processes (and the communication link with them) in the dependency set (e.g. Paxos 2PC [4]).
- Different design, with a knowledge dependency graph in which the critical knowledge state is never blocked by a remote process (e.g. 3PC).

Another use of the message-chain theory is, if it is known that there will be a massage chain between two processes, then other information can be piggybacked along the same message chain. The ring-structured communication above is one example. As another example, from the knowledge dependency graph in Figure 1, we know that if a participant votes *Yes*, then there will be a message chain from the coordinator back to the participant for the outcome to be known. This participant-coordinator-participant chain can then be used for other purposes. For example, in systems where disk writes at participants are more expensive than message passing, the redo and undo logs, instead of being written to disk, can be piggybacked along the message chain and still be guaranteed to be available when the outcome action commit_p or abort_p is performed. Optimizations like *Coordinator Log* and *Implicit Yes-Votes* apply this principle.

5.4.3 Other implementation issues

In the discussions earlier, we mentioned the two different roles a coordinator plays in an atomic commit protocol: achieving agreement and ensuring termination. It is therefore quite natural to separate these roles in two coordinators in special circumstance. *Transfer of coordinator* in *linear 2PC* is one such example.

Another interesting issue is how flexible a protocol execution can be. We consider the flexibilities in two abstraction levels.

- Within the same knowledge dependency graph. In the previous section, we discussed different implementations that can be made for the same knowledge dependency graph. Choice of different implementations can be made either statically or dynamically.
- Switching between knowledge dependency graphs. Different design strategies may lead to different knowledge dependency graphs. For example, the knowledge dependency graphs corresponding to presumed nothing 2PC and presumed abort 2PC are different. However, as long as required knowledge is available, an execution can switch from one graph to another.

Our discussion on implementation issues so far is mainly based on the simple knowledge dependency graph in Figure 1. The graph will look more complicated when, for example, additional vote possibilities are possible (*Yes*, *No*, *NoMatter*/*ReadOnly*, *Yes_heuristic*, *Yes_compensate*). The basic principle would still be the same.

6 Related Work

The logic of process knowledge has been used in design and analysis of atomic commit protocols. In protocol design, the main focus has been on achieving agreement, mostly without considering the effect of failure on decision making [6][8][12]. In protocol analysis, the focus has been on proving impossibilities and lower bounds on number of messages [6][7][10][13].

There has not been much work on relating different coordination protocols in one single framework, either for better understanding of the area or flexible runtime incorporation of them. Chrysanthis et al. [2] gave a detailed overview of 2PC variations. We feel our way (design with knowledge dependency graph and implementation of it) of looking at them is more logical and offers us the possibility to achieve other goals such as flexibility (and switching between different protocols).

7 Summery and Future Work

In this paper, we briefly overviewed a general distributed system model and the logic of process knowledge. We then apply this logic as a tool to first formally specify the atomic commitment problem and later on discuss the design and construction of atomic commit protocols. A design starts with a design strategy which leads to a knowledge dependency graph. Different implementations can be applied to the same graph. During the discussions, we mentioned several protocol variations found in the literature. In fact, we believe that all 2PC variations in the literature can be captured within this framework. Finally we mentioned some other interesting issues such as flexibility.

This work is part of the Arctic Beans project, whose primary goal is to support flexibility and adaptability in an open component-based enterprise architecture. We hope the logic of process knowledge could be a useful tool to reason about and design flexible and adaptable coordination protocols within this architecture.

8 References

[1] Bernstein, P. A., V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[2] Chrysanthis, P. K., G. Samaras and Y. J. Al-Houmaily, "Recovery and Performance of Atomic Commit Processing in Distributed Database Systems", In V. Kumar and M. Hsu (eds.), *Recovery Mechanisms in Database Systems*, pp 370-416, Prentice Hall PTR, 1998.

[3] Fagin, R., J. Y. Halpern, Y. Moses and M. Y. Vardi, *Reasoning About Knowledge*, paperback edition, MIT Press, 2003.

[4] Gray, J. and L. Lamport, "Consensus on Transaction Commit", Technical report MSR-TR-2003-96, 2003.

[5] Gray, J. and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, 1993.

[6] Hadzilacos, V., "A Knowledge-Theoretic Analysis of Atomic Commit Protocols", In P*roceedings of 6th ACM Symposium on Principles of Database Systems*, 1987.

[7] Halpern, J. and Y Moses, "Knowledge and Common Knowledge in a Distributed Environment", In *Proceedings of 3rd ACM Symposium on Principles of Distributed Computing*, 1984.

[8] Janssen, W., "Layers as Knowledge Transitions in the Design of Distributed systems", In Proceedings of 1st International Workshop on Tools and Algorithms for Construction and Analysis of System, Lecture Notes in Computer Science, Vol. 1019, pp. 238-263, Springer-Verlag, 1995.

[9] Lampson, B. W. and David B. Lomet, "A New Presumed Commit Optimization for Two Phase Commit", In *Proceedings of 19th International Conference on Very Large Data Bases*, 1993.

[10] Mazer, M. S. and F. H. Lochovsky, "Analyzing Distributed Commitment by Reasoning about Knowledge", Technical Report CRL 90/10, DEC-CRL, 1990.

[11] Mohan, C., Bruce G. Lindsay and Ron Obermarck, "Transaction Management in the R* Distributed Database Management System". ACM Transactions on Database Systems, 11(4), pp 378-396, 1986.

[12] Moses, Y. and O. Kislev, "Knowledge-Oriented Programming", In *Proceedings of 12th ACM Symposium on Principles of Distributed Computing*, 1993.

[13] Neiger, G. and R. A. Bazzi, "Using Knowledge to Optimally Achieve Coordination in Distributed Systems", Theoretical Computer Science, 220 (1), pp 31-65, 1999.

[14] Object Management Group, *CORBA Services Specification*, Chapter 10, Transaction Service Specification, December, 1998.

[15] Samaras, G., K. Britton, A. Citron and C. Mohan, "Two-Phase Commit Optimizations in a Commercial Distributed Environment", *Distributed and Parallel Databases, 3*(4), pp 325-360, 1995.

[16] X/Open Company Ltd., *Distributed Transaction Processing: The XA Specification.* Document number XO/CAE/91/300, 1991.