A Sound and Complete Distributed Algorithm for Distributed Commerce Transactions

Steven P. Ketchpel
Stanford University
Computer Science Department
Stanford, CA 94305
ketchpel@cs.stanford.edu

Hector Garcia-Molina
Stanford University
Computer Science Department
Stanford, CA 94305
hector@cs.stanford.edu

October 23, 1996

Abstract

In situations where self-interested agents are interacting in an environment of distrust, commercial exchanges may be blocked due to a lack of trust. We propose a fully distributed algorithm that each agent may run to provide minimum guarantees about the outcomes of such exchanges. The algorithm is shown in operation on two examples, one feasible and one not, and is proven to be sound and complete. The algorithm is extended to consider situations in which direct trust does exist between certain participants and those with deadlines for completion. Examples and proofs of soundness and completeness are given for these extensions as well.

1 Introduction

Commercial exchanges typically involve a buyer purchasing goods from a seller. In the electronic world, payment mechanisms have been created to ensure the security of a transaction, so that the seller only receives payment if the buyer receives the goods, and vice versa. However, this model presupposes a two party exchange, involving one supplier and one consumer. In certain sales, the value comes from having materials from a variety of sources, collected and put into final form by one or more intermediaries. However, each party has a local view of the transaction, expecting satisfaction or payment if it fulfills its part of the agreement, irrespective of what
others do. Therefore, the global view of the transaction is much more complicated, but in many cases, a careful ordering of the component steps can ensure that certain basic protections are guaranteed to each member of the transaction. Determining the ordering of the steps is itself a difficult process, because no single agent has a global view of the information. However, a distributed process where each participant uses its local information to interact with its neighbors to perform steps of the transaction results in an overall sequence which provides the desired guarantees.

In a distributed commerce transaction (introduced in [3]), a customer is seeking a collection of goods from a variety of sources, not all of which may be known to him. One example of this situation is a distributed computing environment where idle processors sell their free cycles to computationally intensive jobs. Distributed transactions are also relevant to the domain of information gathering, as we will see from the examples throughout the paper. A customer may specify an information request that may be filled by a set of documents from different sources, some of which are not directly accessible to the customer. There are also many information brokers, who may obtain documents from other sources and re-sell them to customers, who may in turn be brokering the documents to someone else. The environment is one of distrust, so that, in general, a customer will not give payment before being certain of receiving the document. Similarly, a source will not provide a document before being certain of receiving payment.

In order to combat the mistrust between individuals, there are certain agents that are trusted by others. Trusted intermediaries might be a shopping mall, Internet service provider, electronic bank, or the like. They act as intermediaries for exchanges, providing guarantees about the successful completion of them. The trusted intermediary receives the document from the source and the money from the customer, then performs the exchange. If one party does not provide its promised piece, the exchange is canceled, with the goods being returned to their original owner. A simple interaction between a customer, a broker, and a source, is shown in Figure 1. In this case, the customer C has an information request, and knows of a broker B that can help with such requests. The customer is ignorant of the ultimate source S that provides the document, but as long as the broker B knows where to find some source, the request can be fulfilled. The exchanges be-
between C and B and between B and S use trusted intermediaries T1 and T2 respectively.

An important additional point is that perhaps no single document satisfies the information need of the customer. In this case, a customer may specify a conjunction of desired documents. Such a customer agrees to pay for one of the mentioned documents (also called a conjunct) only if all of the documents can be obtained. A broker might also be able to recognize a way to decompose an information request into a conjunction of documents, perhaps fulfilling the request even when no single document matches the original request. Conjunctive situations do arise in the real world—for instance, a third party may provide annotations for a source document. The annotations without the original document are not useful, but the document without the annotations is also of questionable value. To protect the financial interests of multiple authors, information may be divided into parts cryptographically so that all of the parts are required to reconstruct the content. Each co-author is able to control the distribution of one piece, thereby guaranteeing that he receives remuneration for each copy of the work that is legitimately distributed. Illegitimate copying, eavesdropping, interception, and dishonest merchants who provide spurious goods are not prevented by the mechanism proposed here. Additional machinery may take care of some of these concerns, but is outside the scope of this paper.

2 Terminology and Outline of the Paper

In previous work [3], we described a representation language for distributed commerce transactions. This language identifies the agents involved in a transactions and divides them into four types:
1. *Customers* have a request that may be fulfilled by the efforts of other agents.

2. *Sources* have the necessary resources to contribute to the fulfillment of the request.

3. *Brokers* act as middlemen that facilitate transactions by having better knowledge or connections enabling them to locate sources that customers cannot find directly. Brokers operate by purchasing the goods controlled by the source and re-selling them to the customer. When a broker is acquiring a document for resale, it may be called a customer for that exchange.

   These three groups, taken together, are known as *principals*. The brokers and sources are collectively known as *providers*.

4. *Trusted intermediaries* are middlemen that act as guarantors for exchanges between two principals. A customer will supply the money, and the provider will supply the document or other goods. When both principals have sent their requisite piece, the trusted intermediary will complete the exchange by forwarding the pieces to their ultimate recipients. If one piece does not arrive in time, the other is returned to the principal that originally sent it.

   The actions that each agent may execute are limited. An agent may *send a document* to another agent, which concludes with the recipient agent knowing the content of the document. An agent may *send money* to another agent, resulting in the sender’s balance being decremented by the amount, and the recipient’s balance being credited by the amount. A principal may make a request of another principal. A customer may *request a document* from a provider, expecting it to be sent to a trusted intermediary; similarly, a provider may *request payment* be sent to the trusted intermediary from a customer. Trusted intermediaries may not make requests, since they are passive parties to an exchange. They do have a comparable action, however, by which they can *notify* a principal that the exchange is lacking only the piece that must be provided by that principal. For instance, in a simple
exchange, the trusted intermediary would receive the money from the customer then notify the provider that when the provider sends the goods, the exchange may be completed.

In order to describe a particular instance of a distributed commerce transaction problem, the following elements must be specified:

- The agents, and the resources (both documents and money) that they control.
- The connectivity of the agents. To specify the connectivity of the agents, each agent must know its own answer to two questions:
  1. Faced with a particular request, to which provider will a customer turn?
  2. Given the desire for two principals to complete an exchange, which trusted intermediary will they select? (N.B. the answers to this second question implicitly answer the questions of who trusts whom.)
- The decomposition of a query into smaller pieces. It may be necessary to divide a query into sub-queries in order to match available resources that will successfully answer the query. Since all of the sub-queries must be answered in order to answer the original query, this situation is known as a conjunctive one. The original query is called the conjunction and each of the sub-queries is a conjunct. Given a particular query, how will an agent break it down into smaller queries? Different agents may choose different decompositions.

The solution to a distributed commerce transaction problem instance is a way for the agents to use the permissible actions in order to move resources from the sources to the customer. A partial order of actions which is undertaken by the agents is called an execution sequence or just a sequence. However, the distrust between agents results in added constraints on these execution sequences. Intuitively, we want to prevent agents from being in a position where they might be cheated by a malicious, untrusted agent. The solution to the problem must be safe for each agent, where a sequence is safe for one agent if:
1. When the agent acts as a customer, it never spends money without being guaranteed of receiving the promised document in return.

2. When the agent acts as a provider, it never sends a document to a customer without being guaranteed of receiving payment.

3. When the agent acts as a customer with a conjunctive request, it will never pay for one document unless it is able to obtain all of the conjuncts. (This risk is known as “buying half a conjunction.”)

4. When the agent acts as a broker, it will never purchase a document unless it is guaranteed that a customer will re-purchase the document.

Each of the agents involved in the transaction must be convinced that all of the actions it performs can lead only to sequences which are safe for it, even if other principals deviate from the expected sequence. In the event that each agent considers the sequence safe, the sequence is called riskless. Trusted agents are trusted to follow the sequence.

The goal of this paper is to describe a distributed algorithm which can, for any solvable instance of a distributed commerce transaction problem, generate a riskless sequence of steps that results in the customer’s request being fulfilled if all the agents follow the generated sequence. Such an instance is called feasible. Other instances are infeasible—there is no execution sequence which is safe for all of the agents. (Although infeasible transactions may be transformed into feasible ones if indemnities, money that is given to a trusted intermediary that will be forfeited if certain conditions are not met, or returned if the conditions are met. Indemnities were introduced in [3], and are not discussed further here.)

The actions will frequently be grouped into larger units for reference. One such unit is the pair-wise exchange which is the sequence of actions for a customer to receive the goods from a given provider and that provider to receive the money. The global exchange is the complete set of pair-wise exchanges that form the solution execution sequence to the distributed commerce transaction problem instance. Where exchange is used without either qualifier, the meaning should be clear from context.
For the example in Figure 1, one riskless sequence of operations requires customer C to give the money for the document to trusted intermediary T1, enabling broker B to purchase the desired document from source S, confident that the customer will not be able to back out of the purchase. In order to protect itself from an untrusted source, broker B completes the pair-wise exchange with the source through trusted intermediary T2. The trusted intermediary T1 guarantees that when the broker B provides the document, the money that the customer has entrusted to it will be paid to B, with C receiving the document at the same time.

Finding a riskless execution sequence is more difficult when the request is part of a conjunction. In a conjunctive case, the customer is unwilling to give money to the brokers in charge of each conjunct, in case one or more of the conjuncts is unavailable, and the customer ends up paying for an unusable subset of the desired documents.

The sequencing graph formalism introduced in [3] used a graphical representation of distributed commerce transactions to determine whether a riskless execution ordering existed. That algorithm makes use of a series of graph reduction steps, applied to a representation of the full problem instance, information that is available only if all the agents combine their information. An individual agent does not have enough information to decide what steps are appropriate without receiving instructions from a centralized entity.

The algorithm for controlling the behavior of the agents that we describe here is fully distributed, so that each agent is able to plan and carry out its actions based on its own state and the requests it receives from neighboring agents. We further show that this approach is *sound*, so that if the algorithm runs to completion, the proposed execution sequence is riskless, and even if other agents deviate from the prescribed sequence, each agent will be safe. The algorithm described here is also proven to be *complete*. That is, if there is some riskless sequence of the permitted actions that results in a completed exchange for a particular scenario, the algorithm will also find a riskless execution sequence.

In Section 3, we give an overview of the algorithm. The success of the algorithm relies upon several assumptions, which we detail in Section 4. A more detailed description of the algorithm and its behavior on an example
follows in Section 5. Section 6 contains a more complicated example, showing the conjunctive case. Sections 7 and 8 prove the algorithm’s soundness and completeness. Section 9 shows how to handle cases where principals will interact directly with another principal rather than relying on a trusted intermediary. In that case we say that one principal has direct trust for another. Direct trust need not be symmetric. Section 10 considers exchanges where time is important and they must be completed by a deadline. In that section, we develop a temporal version of the algorithm from Section 5, which for the sake of contrast, we call atemporal. A few optimizations are suggested in Section 11, and we conclude with a summary, implications, and directions for future work in Section 12. The full algorithm appears in the Appendix.

3 Algorithm Overview

This section describes at a high conceptual level the operation of the fully distributed algorithm for finding riskless execution sequences in distributed transactions. The following section develops the algorithm in more detail, working through an example of a simple transaction. A full specification appears in the pseudocode presented in the Appendix.

Each agent operates as an autonomous processor, with its own knowledge limited to the documents for which it serves as a source, and the areas of expertise for agents with which it can communicate directly. These data, along with the customer’s information request, form the inputs to the algorithm. The output is the sequence of steps which is executed by the agents in the system. This sequence will fulfill (if possible) the customer’s request, while still ensuring that no agent risks entering an unsafe state where it has paid for something it did not receive or was not paid for something it sent. If the algorithm is unable to find a riskless sequence, then none exists.

When one agent has an information request, it is sent to appropriate neighboring agents who either fulfill the request or redistribute it to other sources that might not be directly available to the original customer. Several pair-wise exchanges may then be required to move the information back to the ultimate customer. The status of each of these pair-wise exchanges is kept locally by the agents involved, and is updated as goods and money flow back and forth. The agents are “event-driven”, reacting to events that are
incoming messages describing customer requests, notifications from trusted intermediaries, or the delivery of documents or money. These messages are represented as method invocations by the sender upon the recipient agent.

Figure 2 shows an example of a distributed commerce transaction where a customer contacts a broker who decomposes the request into two sub-queries that are sent to two different sources. This figure is overlaid with a representation of the different process spaces, showing the computational boundaries between the agents and which pieces of the state of the global exchange are available to each agent.

The simplest description of the agents in a distributed commerce transaction is as event processing loops. Their operation follows a simple cycle:

1. Wait for a message event (request, notify, delivery) to arrive.
2. Update the local state concerning the pair-wise exchange with the message sender, based on the content of the message.
3. Based on this new local state, dispatch the next action(s).
The state update of Step 2 is deterministic and always well-defined. It records whether a document has been requested, sent, or received, and whether the corresponding payment has been requested, sent, or received. In Step 3, the recipient agent considers the new state of this pair-wise exchange, in combination with the state of other pair-wise exchanges that might be part of the same conjunction as the newly updated exchange. As a result of this consideration, the agent decides which actions to take next. There are five cases that must be separated (described in greater detail in the function check_for_next):

1. *The exchange cannot be completed in time.* This case is discussed in Section 10.

2. *The agent has all of the pieces to fulfill the information request.* In this case, the agent is either done (if it is the ultimate customer) or can send the answer on to the requester.

In the example of Figure 2, if the broker had already received the document from Source 2, then when it receives the document from Source 1, B has all of the necessary components to fulfill the request from C. Since B is not the ultimate customer (C is), B will send them on to C.

3. *The agent has promises from trusted intermediaries that they have the documents that will fulfill the request.* In this case, the agent can safely send money to the trusted intermediaries to obtain the documents, reducing the problem to the previous case.

In the example of Figure 2, the broker will request the documents from Sources 1 and 2. The sources have nothing to lose by sending them to the trusted intermediaries that would facilitate the exchange between themselves and the broker. Therefore, when the trusted intermediaries (not shown in this figure) receive the requested goods, they send promises to the broker that when the broker sends money, it can be guaranteed to receive the desired documents in exchange. When all the promises arrive at the broker, case 3 of check_for_next occurs.
4. The agent and trusted intermediaries are still missing one piece for the request. In this case, the agent guarantees payment to the source of the last piece, if its customer has done likewise. This may expedite the acquisition of the missing piece.

In the example of Figure 2, if the broker has made requests of the two sources, but one has decided not to comply immediately, then this case will arise. The broker will receive notification from one of the trusted intermediaries saying that the one document is available. However, without the second document, the first is of no value. Therefore, the broker undertakes more effort to acquire the second document (case 4 of check_for_next). By sending money to the trusted intermediary, the broker demonstrates its good faith without putting itself at any risk.

5. The agents and trusted intermediaries are missing two or more pieces for the request, or the customer has not guaranteed payment. In this case, the agent is reduced to merely asking sources to provide the necessary pieces without guarantee of payment. The sources may be willing to do so if they are not required to spend any resources. If they have to spend money to acquire these documents from other sources, however, they will not without a binding promise of payment.

In the example of Figure 2, the broker is not willing to spend money for the documents from the two sources unless it is sure that it will obtain both. Therefore, rather than sending money, it first makes a request, asking one or both of the sources to forward the desired document to the shared trusted intermediary in order to move the exchange along.

4 Assumptions

Several assumptions are necessary in order to make the search for riskless execution sequences tractable. In some cases, we simply assume that related problems of distrust are outside the scope of our concern and can be attacked separately by other technical solutions. For example, a source providing a
document to an untrusted customer is taking on multiple risks: the customer may not pay for the document, depriving the source of income for one sale; or the customer may make illegal copies of the document, distributing them for free to other potential customers of the source. The algorithm we describe here is designed to solve the first problem, but does nothing to prevent the second. Therefore, we assume that agents in our system will not display this type of undesirable behavior, or there are mechanisms in place to prevent or detect illegal duplicates [5, 1]. The flip side of this assumption is that sources will also be trustworthy in certain respects. A provider may promise a document which satisfies the customer’s request, but instead may actually deliver a piece of meaningless garbage. Although trusted intermediaries may be used to help police this problem [2], we simply assume that there is no mismatch between what the source promises and what it provides.

The previous two assumptions have simply to do with the scope of the problem we are considering, and do not impact the correctness of our solution, because of the way we define the problem. The remaining assumptions have to do with certain properties of the problem instance that simplify our solution.

First, we assume that any pair of principals that wishes to make a pair-wise exchange has access to a shared trusted intermediary. Until Section 9, we will further assume that two principals always make pair-wise exchanges through a trusted intermediary. Second, we assume that the price of any single document is negligible compared to the resources of an agent. This assumption implies that a broker will always be able to pay “out-of-pocket” to acquire a document without relying on its customer’s payment in order to purchase the document from the source. Third, we ignore the complexity of a customer making the same request of multiple providers, planning to pay whichever responds first, or offers the lowest price. In our formulation, no customer ever makes the same request to more than one source. Fourth, we assume that no messages are lost in transit. Relaxing these assumptions offers opportunities for future research.

Finally, in the description of the algorithm in the text and in the Appendix, we assume that parameters to method calls are passed by reference rather than by value, and that the callee is able to modify certain values in data structures that were created by the caller. In an atmosphere of
mistrust between agents, this assumption is unrealistic. However, we make the assumption to simplify our explanation of the algorithm. Section 5.3 discusses the complications that would be required for the more realistic call-by-value method invocations.

5 Algorithm Details

This section describes in greater detail the operations of the search for a safe execution sequence for a distributed commerce transaction. The algorithm is run by each agent and leads only to safe execution sequences for that agent. If there is a riskless sequence which will satisfy the customer’s request, the algorithm will find it and execute it. Agents who follow the protocol do not risk loss even if others deviate from the protocol, provided that trusted agents obey the algorithm. The data structures used by the algorithm and the algorithm itself will be described, and they form the major contribution of this paper. Since the inner workings of the algorithm are somewhat hard to follow in the abstract, we will first show the basics as we walk through a simple example from Figure 1 of a consumer C ordering a document D from broker B, who in turn acquires it from source S. The consumer and broker make their exchange through trusted intermediary T1, while the broker and the source use trusted intermediary T2 for their exchange.

The rest of this section shows how the algorithm generates each of the main steps for this example:

1. C’s setting up the request at C.
2. C’s communicating the request to T1 and sending C’s money to T1.
3. T1’s notifying B of the request and the presence of C’s money.
4. B’s communicating the request to T2 and sending B’s money to T2.
5. T2’s notifying S of the request and the presence of B’s money.
6. S’s sending the document to T2.
7. T2’s sending the document to B and B’s money to S.
8. B’s sending the document to T1.
9. T1’s sending the document to C and C’s money to B.

These nine steps form the structure for the remainder of this section. Each step is the header for a subsection that shows the particular operations undertaken for that task and the instantiated data structures for this example. The complete algorithmic description for the procedures mentioned may be found in the Appendix.

5.1 C’s setting up the request at C.

In order to initiate the exchange, the consumer creates a TaskRecord that will store the details of the exchange between two principals C and B, as shown in Figure 3, with a description of the fields in the Appendix. The first components of the TaskRecord (also abbreviated TR) are the agents and the document involved in the exchange, in this case C is obtaining D from B through T1. Since C knows about the areas of expertise of its immediate neighbors, C is able to select B as the most relevant to handle a request for D. C further knows that T1 is the trusted intermediary that it should use for transactions with B. This TaskRecord will also be updated as the exchange continues, reflecting the current status of the transaction.

Two status variables are used in each TR: one to record the status of the document, the other, the payment. For instance, the document’s status might be RECEIVED, indicating that C has received D from T1. Other possible values are: SENT, so that B has sent D to T1, but C has not yet gotten it; REQUESTED, C has requested that B send D, but T1 has not received it yet; NOTHING, where C has made no action with respect to D and B; and EXPIRED, where an exchange has not been completed in time, so C returns the document it had acquired from B. The status of the payment for the document may take on the same values, though the polarity is reversed, so that it is B who requests the money, and C that sends it to T1. At the start of our example, there has been no action with respect to either the document or the money, so the status of both is NOTHING. The TaskRecord constructed so far (and more detail will be added later), then, is1:

\[\text{T}1\]

Normally, only the document ID and client agent would be filled out at this stage. The remaining fields are filled in later during \texttt{AcquireSet}. This liberty was taken for expository purposes.
Figure 3: (a) The structure of a TaskRecord. (b) A partially instantiated TaskRecord for one step of Example 1.
<D, C, T1, B, NOTHING, NOTHING>

However, this TaskRecord does not record all of the information that C has about the exchange. C wants the document D, and has decided to purchase it, consciously setting aside funds for it. In order to represent this commitment, we essentially model a separate exchange between two personas of C, one that wants the document, and the one that has the money. A new TR is created, with C playing the role of both client and source, with no trusted intermediary in the middle. By making each agent both a customer and a source, the algorithm is able to treat all of the agents in a uniform matter. In this special case, the banker persona has sent the money while the other persona has requested the document. Therefore, the money has been committed (SENT) and the document status is REQUESTED.

< D, C, , C, REQUESTED, SENT>

There is an important connection between the two tasks represented in the two TaskRecords we have shown: the first between C and B is done in order to provide the document needed for the second between C and itself. Since these two tasks are related, with the second being a “client” or consumer of the document, while the first is the way that the document is provided or its “source”, we create a new data structure called a SetRecord which has a “clientTR” and a set of “SourceTRs”, which are given the values of these two TR’s.

ClientTR:  <D, C, , C, REQUESTED, SENT>

SourceTRs:  {<D, C, T1, B, NOTHING, NOTHING>}

The remaining pieces of the TaskRecord and SetRecords (other than those used for deadlines, discussed in Section 10) are pointers which allow the construction of chains of exchanges that are necessary for the traversal of these data structures during a distributed transaction. Each SetRecord is used to acquire documents from all of the sources specified in the SourceTRs, and send them out to the client agent of the clientTR. This acquisition and subsequent re-sending may be in support of a higher level request, for instance, when broker B acquires D from S, it does so in service of the higher level request of C to B. That dependency would be recorded in the SetRecord as a pointer from B’s SetRecord to C’s SetRecord, its parent. The SR we are constructing here, however, is a top-level request, and C is the ultimate customer. Therefore, the Parent field of the SR is null. Each TR
also maintains a pointer to allow the traversal of these SetRecord chains. A client TR has a pointer to its containing SR. The SourceTRs’ however, point to the SRs that are constructed by the source providing the document, a step which will become clearer later in the execution of the algorithm. When each SourceTR is first constructed, however, the source’s SR is unknown, so a null pointer is used. The complete (for all but deadlines introduced in Section 10) SetRecord that C constructs is:

\[
SR_C = \text{Parent: } \emptyset \\
\text{ClientTR: } \langle D, C, \emptyset, C, \text{REQUESTED, SENT, } SR_C \rangle \\
\text{SourceTRs: } \{\langle D, C, T1, B, \text{NOTHING, NOTHING, } \emptyset \rangle\}
\]

Figure 4 is a generic SR with two SourceTRs. A graphical representation of SR\(_C\) appears in Figure 5.

### 5.2 C’s communicating the request and sending C’s money to T1.

Having constructed this SR, C calls the `AcquireSet` function, which functions as the entry point to the algorithm, using \(SR_C\) as a parameter. The purpose of `AcquireSet` is to figure out which of the requested documents the agent already has and where to get the rest. This function’s name, and the name of the SetRecord itself, recognize that a consumer may want a set of documents, or a broker may be able to fulfill an information request only by decomposing it into documents to be obtained from several sources. In either case, the `SourceTRs` field of a SetRecord needs the ability to hold multiple documents. It is, therefore, a set of TR’s, all of which must be acquired.

The operation of `AcquireSet` is relatively straightforward. Since C does not have \(D\), C plans a way to obtain \(D\), and fills in the fields of the TR in `SourceTRs`. (The values of these fields were shown in the previous section for expository purposes, even though they would not usually be filled in until this stage of `AcquireSet`.)

At the conclusion of `AcquireSet`, the function `checkfornext` is called. This function is an important dispatch function which determines what process is enabled by the current status of the transaction. It is essentially a single `case` statement that tests several conditions and undertakes the next appropriate safe action. From a high level, there are five cases which the
SETRECORD

<table>
<thead>
<tr>
<th>CLIENT TR</th>
<th>SOURCE TR #1</th>
<th>SOURCE TR #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client Agent</td>
<td>Client Agent</td>
<td>Client Agent</td>
</tr>
<tr>
<td>Trusted Int.</td>
<td>Trusted Int.</td>
<td>Trusted Int.</td>
</tr>
<tr>
<td>Source Agent</td>
<td>Source Agent</td>
<td>Source Agent</td>
</tr>
<tr>
<td>Deadline</td>
<td>Deadline</td>
<td>Deadline</td>
</tr>
<tr>
<td>Ult. Deadline</td>
<td>Ult. Deadline</td>
<td>Ult. Deadline</td>
</tr>
<tr>
<td>DocStatus</td>
<td>DocStatus</td>
<td>DocStatus</td>
</tr>
<tr>
<td>MoneyStatus</td>
<td>MoneyStatus</td>
<td>MoneyStatus</td>
</tr>
<tr>
<td>SR Pointer</td>
<td>SR Pointer</td>
<td>SR Pointer</td>
</tr>
</tbody>
</table>

Figure 4: The structure of a SetRecord and its component TaskRecords.
routine distinguishes among:

1. The exchange cannot be completed before an established deadline (see Section 10).

2. The agent has acquired all of the desired documents in the set of source TRs. The appropriate action is to send them to the client, or trusted intermediary shared with the client.

3. Trusted intermediaries hold all of the desired documents that the agent does not have yet. The agent can merely send payment and receive all the needed documents.²

4. Trusted intermediaries hold all but one of the documents that the agent does not have yet. Here, the agent should offer payment to the trusted intermediary (if it can³) to obtain the missing document. If the last document is obtained, the customer can acquire the other documents from the trusted intermediaries. Otherwise, the customer will receive a refund.

² Actually, this is oversimplified a bit. If the agent’s client has not sent money yet, then the agent cannot safely pay for the documents, but must wait for a trusted intermediary to receive the customer’s payment.

³ Once again, payment can only be sent to the source if payment has been guaranteed by the customer (i.e., received by the trusted intermediary.)
5. Multiple documents are not yet at the trusted intermediaries. If the documents have not yet been requested, the agent should request them. Otherwise, it must wait for other agents to provide the documents.

When check_for_next is called on SRC, the fourth case is selected, since one document, D, has not been acquired or requested (SR.SourceTRs.DocStatus = NOTHING). This fourth clause of check_for_next calls Send_Payment_or_Request to obtain the missing document.

In this example, D is quickly identified as the missing document (it is the only TR in SourceTRs), and the client’s MoneyStatus is known to be SENT (because C’s “banker persona” has allocated the money for the purchase of D). Therefore, C is able to send its money to T1, the trusted intermediary of the SourceTR. The method Send_Payment_or_Request terminates by invoking the Receive_Payment method on trusted intermediary T1. In order for T1 to properly handle the payment, it needs the information which is specific to the payment (e.g., the account number) and the document which is being paid for. In this case, as T1 executes Receive_Payment, it uses the information stored in the client’s SetRecord4 which shows both the source and the destination of this exchange, specifically in this example, SRC.

T1 stores locally the receipt of payment and saves the payment for future delivery to B, when B provides D.

5.3 T1’s notifying B of the request, and the presence of C’s money.

At this stage, T1 concludes Receive_Payment by signaling B via the Notify_Payment call that B will receive money as soon as D is provided. Since this request is a new one to B, it creates a new SetRecord. The difference between call-by-reference and call-by-value parameter passing becomes significant at this point. Figure 2 showed the call-by-value scheme, where each agent had its own copy of the information related to a single pairwise exchange. Alternatively, in a call-by-reference formulation, there would be a single copy of

4 Here a trusted intermediary is being given permission to read another agent’s local storage. In practice, this would not be permissible, but instead would be passed as a parameter to the intermediary. The algorithm can be easily extended to accomplish this, but it needlessly complicates the description of the situation. Further discussion appears in Section 5.3.
the information located where both agents could reference and modify it. A single copy is sufficient because, as our intuition suggests, since the document is being provided by one agent to another, the "out-box" of the source is the "in-box" of the client. Therefore, the clientTR of B’s SR is exactly the same as the SourceTR from C, which has already been created.

However, at the time that C created it, the value of the SRPointer (which should point to B’s SR) was unknown, since B’s SR did not even exist at the time that C created the TR. Under the call-by-reference formulation, B fills in the pointer to its SR in the SRPointer field of its clientTR, and since that record is identical (shares the same memory locations) with C’s SourceTR, that record is implicitly updated at the same time. In the call-by-value formulation, B updates its own clientTR with the value for the SRPointer, but that operation has no effect on C’s sourceTR. Therefore, B must pass a message to C with the necessary value so that C can update its sourceTR. The method for this transfer is not covered in the Appendix, which for simplicity assumes a call-by-reference model.

In order to complete B’s SetRecord, the Parent field must be filled in, which is done in B’s NotifyPayment routine as the one step of constructing the SR. Since B undertook this acquisition to fulfill C’s request, C’s SR is the parent of B’s, hence the need to pass C’s SetRecord in the ReceivePayment and NotifyPayment calls.

The last step that NotifyPayment performs in the context of creating the SR is to break C’s request into documents that B will be able to obtain. In this example, since B knows that S can directly provide D, this step is trivial, and no decomposition is required. Decompose does create a new TR, with the desired document set to D, and the broker itself as the client. So, the SetRecord created by B is:

\[
\text{SR}_B = \begin{array}{l}
\text{Parent: } \text{SR}_C \\
\text{ClientTR: } <D, C, T1, B, \text{NOTHING}, \text{SENT}, \text{SR}_B > \\
\text{SourceTRs: } \{<D, B, \_ , \_ , \_ , \_ , \emptyset >\}
\end{array}
\]

The blank fields of the SourceTR are filled in by AcquireSet. B knows that the relevant source for D is S, and the trusted intermediary of choice for transactions with S is T2. Therefore, when the AcquireSet is ready to call checkfornext, the value of SR_B is
Figure 6: SetRecords for a customer and his broker.

$$SR_B = \text{Parent: } SR_C$$

$$\text{ClientTR: } \langle D, C, T_1, B, \text{NOTHING, SENT, } SR_B \rangle$$

$$\text{SourceTRs: } \{ \langle D, B, T_2, S, \text{NOTHING, NOTHING, } 0 \rangle \}$$

The relationship between $SR_B$ and $SR_C$ is shown in Figure 6.

5.4 B’s communicating the request and sending B’s money to T2.

The last line of agent B’s Acquire_Set function calls check_for_next on this SetRecord. Again, the current SR falls into the domain of case four of check_for_next. Because C has guaranteed its intention to purchase by giving money to T1, B can safely give its own money to T2. B is confident that when S provides the desired document, B will be able to get its money back by giving the document to T1. So when Send_Payment_or_Request is called with this $SR_B$, D is again selected as the missing document. With C’s money at T1, the MoneyStatus variable for the exchange between C and B has the value SENT, and B sends its payment to T2 by invoking the Receive_Payment method on the trusted intermediary T2.

T2, as T1 did before it, checks to see if this payment completes an exchange. Finding that it does not, T2 uses the Notify_Payment method on S to continue the transaction.
T2’s notifying S of the request and the presence of B’s money.

Notify_Payment causes S to create a new SR, which we call SR_S, with 
SR_B as its Parent, the incoming TR as its clientTR, and a new, incomplete 
SourceTR showing that document D is being requested. This new SR, called 
SR_S, is passed to Acquire_Set.

\[
\begin{align*}
SR_S & = \text{Parent: } SR_B \\
\text{ClientTR: } & <D, B, T2, S, \text{NOTHING, SENT, SR_C} > \\
\text{SourceTRs: } & \{<D, S, \_ , \_ , \_ , \_ , \_ , \emptyset >\}
\end{align*}
\]

As S is executing Acquire_Set, it finds that it has the document in question. Therefore, the finalized SourceTR (modeling the exchange between 
two personas of S) is \(<D, S, \emptyset, S, \text{RECEIVED, \_ , \_ , \_ , \emptyset} >\). The relationship 
among the three SR’s is shown in Figure 7. When check_for_next is called 
on SR_S, the second clause (showing that all documents have been received) 
is activated. Since the ClientAgent of the ClientTR is B and not S itself (i.e., 
S is not the final customer), the documents must be sent to their customer, 
and Send_Doc is invoked on T2, the trusted intermediary that B shares with 
S, passing the document and SR_S as parameters.

S’s sending the document to T2.

The process for sending a document to another agent, Send_Doc is quite simple. We will add more functionality later with direct trust between parties, 
but the main purpose will be the same: update the status in the TR and 
invoke the Receive_Doc method of the trusted intermediary with the text 
of the document (the result of composing the documents from the sources) 
and the SetRecord. In this case, T2 is the target agent of the Receive_Doc 
call.

T2’s sending the document to B, B’s money to S.

The trusted intermediary’s Receive_Doc routine is completely symmetric 
to Receive_Payment. Since T2 already has the payment for this exchange 
from B, it will complete the exchange, sending the document to the broker 
B and payment to Source S. T2 invokes the method Receive_Payment on S, 
which enables S to deposit the cash, update the status of the TR, and call
the `check_for_next` dispatching function. The call to `check_for_next` ends processing at S, since none of the clauses is activated. T2 accomplishes the second half of the exchange by sending the document to B, which it does by invoking the `Receive_Doc` method on B. This method is comparable to `Receive_Payment`, in that it stores the document and calls `check_for_next`.

### 5.8 B’s sending the document to T1.

When B evaluates the conditions of `check_for_next`, it finds that the second clause is applicable, because all of the documents (in this case, just D) have been received. Since B is not the ultimate customer (SR.client.ClientAgent is C), B calls its own method `Send_Doc`, which in turn calls the `receive_Doc` method on trusted intermediary T1.

### 5.9 T1’s sending the document to C, the money to B.

T1 sees a situation similar to the one T2 faced for the previous exchange, and continues it in the same way—`Receive_Payment` is invoked on B, and `Receive_Doc` on C. Both invocations result in status updates and calls to `check_for_next`. B finds that none of the clauses of `check_for_next` is warranted, so B is done with this transaction. Upon C’s execution of `check_for_next`, it discovers that the second clause is activated, but since C is the ultimate customer (SR.client.ClientAgent = C), rather than sending the document on, the success method is called, and the transaction is complete at all of the sites, concluding the example.

### 6 Conjunctive Example

In the second example (Figure 8), customer C desires two documents (D1 and D2) and must interact with brokers B1 and B2 in order to get them from S1 and S2. T1 is the trusted intermediary between C and B1, T2 is for C to B2, T3 is B1 to S1, and T4 is B2 to S2. The set of SRs that the algorithm generates for this exchange is shown in Figure 9. This example is infeasible without indemnities, so the algorithm will not find a safe execution ordering. Again, temporal issues are deferred to a later section.
Figure 7: The partial contents of the SetRecords for Example 1.

Figure 8: The parties involved in Example 2.
Figure 9: The contents of the SetRecords for Example 2.

The initial SetRecord is similar to that of the first example in that it has a clientTR showing a transfer between two personas of the same agent. We do see a difference however, in that the target information request may only be fulfilled by a conjunction of two components, D1 and D2. Since these documents are coming from different sources, there are two different TaskRecords in the SourceTRs set. C’s SetRecord is shown here:

\[ \text{SR}_C = \begin{array}{l}
\text{Parent: } \emptyset \\
\text{Client: } <\text{D1} \land \text{D2}, \text{C}, \emptyset, \text{C}, \text{NOTHING}, \text{SENT}, \text{SR}_C > \\
\text{SourceTRs: } \{<\text{D1}, \text{C}, -, -, -, -, \emptyset>,<\text{D2}, \text{C}, -, -, -, -, \emptyset>\}
\end{array} \]

When Acquire_Set is passed this SetRecord, it fills in the information relevant to finding the component documents, yielding TaskRecords:

\[ \text{SourceTRs: } \{<\text{D1}, \text{C}, \text{T1}, \text{B1}, \text{NOTHING}, \text{NOTHING}, \emptyset>, <\text{D2}, \text{C}, \text{T2}, \text{B2}, \text{NOTHING}, \text{NOTHING}, \emptyset>\} \]

When this SetRecord is passed to check_for_next, only the fifth clause is activated. Since there are two documents that are not yet at the trusted intermediary, C is unable to advance payment. C faces the risk of "buying
half a conjunction”—for example B1 providing document D1 while B2 gives up, and returns the payment. In that case, C has spent half of its money but not obtained the desired set of documents. So, instead, C performs the riskless action of requesting both documents from the respective brokers, and the SR is updated to reflect the new DocStatus values:

SourceTRs: \{<D1, C, T1, B1, REQUESTED, NOTHING, 0> <D2, C, T2, B2, REQUESTED, NOTHING, 0>\}

The processing performed by B1 and B2 is exactly symmetric, and the operations may be performed in parallel, or interleaved in some way unknown to C. Since there are no interactions, we will assume that B1 processes its request first. The first step that RequestDoc undertakes is to create a new SetRecord. The agent B1 uses C’s TR as the client TaskRecord, updating its local copy (and C’s if this is necessary as a separate step) with a SRPointer to the newly created SetRecord. Since C has already performed a decomposition requesting only D1 from B1, no further decomposition is performed, and the only SourceTRs value is the partially described request for D1.

\[SR_{B1} = \text{Parent}: SR_C\]
\[\text{ClientTR}: <D1, C, T1, B1, REQUESTED, NOTHING, SR_{B1}>\]
\[\text{SourceTRs}: \{<D1, B1, _, _, _, _, 0>\}\]

AcquireSet fleshes out the TR for the exchange between B1 and S1, yielding:

\[SR_{B1} = \text{Parent}: SR_C\]
\[\text{ClientTR}: <D1, C, T1, B1, REQUESTED, NOTHING, SR_{B1}>\]
\[\text{SourceTRs}: \{<D1, B1, T3, S1, NOTHING, NOTHING, 0>\}\]

This completed SR is passed to check for next. Although there is a single SourceTR which has not yet been received, the fourth clause is not applicable, because the broker has not received payment from C (SR_{B1}.Client.MoneyStatus is NOTHING). Therefore, control falls through to the fifth clause, where B1 requests D1 from S1. When B1 executes the RequestDoc method on source S1, a new SetRecord is created by S1, which we call SR_{S1}.

\[SR_{S1} = \text{Parent}: SR_{B1}\]
\[\text{ClientTR}: <D1, B1, T3, S1, REQUESTED, NOTHING, SR_{S1}>\]
\[\text{SourceTRs}: \{<D1, B1, T3, S1, REQUESTED, NOTHING, 0>\}\]

But when S1’s AcquireSet receives this SetRecord, it recognizes that S1 has the desired document D1. Therefore, the TR is filled out showing a
completed exchange from S1 to S1, using no trusted intermediary, with the
document having been RECEIVED:

\[ SR_{S1} = \text{Parent: } SR_{B1} \]

\[ \text{ClientTR: } <D1, B1, T3, S1, REQUESTED, NOTHING, SR_{S1}> \]

\[ \text{SourceTRs: } \{<D1, S1, \emptyset, S1, RECEIVED, NOTHING, \emptyset>\} \]

When \text{check\_for\_next} receives \( SR_{S1} \), all of the desired documents listed
in \text{SourceTRs} are available, so the second clause is applicable. S1 inter-
acts with B1 through T3 (which is \( SR_{S1}.\text{client}.\text{TrustedIntermediary} \)),
so when \text{Send\_Doc} is called, the immediate effect is to call \text{Receive\_Doc} on
T3, using the document as one parameter, the TR between S1 and B1 as
the second, and B1's SR as the third. T3 checks for whether this new arrival
completes an exchange, but finds instead that no money has yet been com-
mitted for this document, so T3 notifies the client B1 that the document
has been received, and the exchange can be completed as soon as the money
is received. T3 achieves this by invoking the \text{Notify\_Doc} method on B1.
When B1 executes \text{Notify\_Doc}, it updates the DocStatus to SENT, and
calls \text{check\_for\_next}.

When B1 calls \text{check\_for\_next}, the third clause is activated, since all of
the desired documents in the SourceTRs set (just D1) have already been sent
to the trusted intermediary, and consequently have the DocStatus “SENT”.
The third clause invokes a new routine, \text{Send\_All\_Payments\_or\_Request},
with B1's SR as the parameter.

B1 sees whether C has guaranteed payment for this document by sending
money to their trusted intermediary, but C has not, since facing the risk of
buying half a conjunction it could not commit payment for either piece. This
is reflected in the \( SR_{S1}.\text{client}.\text{MoneyStatus} \) of NOTHING. Without funds
committed from C, B1 is unable in turn to extend money to S1, in case
C decides to retract its offer. Therefore, \text{Send\_All\_Payments\_or\_Request}
cannot send money, and this branch of the transaction blocks until payment
is received from C. (An enhancement to the algorithm allows improved effi-
ciency. See the details of \text{Request\_Payment}, in Section 11.3.)

On the other branch of the transaction, going from C to B2 to S2, a
completely symmetric sequence of steps is occurring, yielding the same re-
sult: D2 is transferred to T4, but in spite of notification of this fact, B2
cannot transfer funds to pay for it. As in the other branch, completion of
the exchange is blocked, and the global exchange cannot be completed.

7 Soundness

In this section, we prove that the algorithm described in Section 5 is sound, that is, any sequence of actions that the algorithm generates is safe for all of the participating agents. To review, a sequence is safe for an agent if all of the following conditions are met:

1. When the agent acts as a customer, it never spends money without being guaranteed of receiving the promised document in return.

2. When the agent acts as a provider, it never sends a document to a customer without being guaranteed of receiving payment.

3. When the agent acts as a customer with a conjunctive request, it will never pay for one document unless it is able to obtain all of the conjuncts.

4. When the agent acts as a broker, it will never purchase a document unless it is guaranteed that a customer will re-purchase the document.

Theorem 7.1: Any sequence of actions produced by the algorithm is riskless.

Proof:

We show in turn that each of the four unsafe conditions cannot arise in a sequence of actions generated by the algorithm.

1. Whenever a customer sends money, it always receives the document or a refund: A customer only gives money to trusted intermediaries (in either SendAllPayments or Request or SendPayment or Request). The trusted intermediary will refund the customer’s money or send it to the provider in conjunction with sending the document to the customer. Therefore, any sequence the algorithm generates will ensure that whenever a customer spends money without having it refunded, the customer will get the requested document.

2. Whenever a provider sends a document, it always receives payment or the document is returned: Similar to the previous case. Providers only
give documents to trusted intermediaries that send them on to the customers only when the payment can be sent to the provider at the same time.

3. **A customer never buys half a conjunction**: A customer never sends money for one conjunct unless it is sure that all of the other conjuncts are readily available. Payments are made only when at most one document is not held by the customer or an intermediary the customer trusts (check for next case three or four).

After payment for the last missing conjunct is forwarded to the shared trusted intermediary, there are two possible outcomes:

(a) The provider can deliver the document, allowing the customer to obtain the full conjunction by sending payment to each trusted intermediary, or

(b) The provider fails to deliver the document, so the trusted intermediary returns the customer's payment, and the customer has lost nothing, since it has not paid for any of the other conjuncts.

4. **A broker never buys a document without being able to re-sell it**: Here again, a conservative policy by the broker eliminates its risk to this type of exposure. The only condition under which a broker will purchase a document for re-sale is if the MoneyStatus of the TaskRecord for its client is SENT. The only way this status is recorded at the broker is after a NotifyPayment is received from the trusted intermediary, which is contingent upon the trusted intermediary's having received the payment. Therefore, if the broker does obtain the document, it will be able to re-sell the document, since the trusted intermediary holds the money from the broker's intended customer, ensuring the customer cannot back out of the purchase.

Therefore, if none of the four unsafe cases may arise from the sequences suggested by the algorithm, each agent finds the sequence safe, so it is riskless for the population as a whole. Consequently, any sequence suggested by the algorithm is riskless. ■
8 Completeness

In this section, we establish that the algorithm developed in Section 5 is complete, that is, it will find a solution whenever one exists. In particular, we will show that if there is a safe execution sequence for a scenario which results in the customer obtaining the desired documents, the algorithm will find it or one that works equivalently well. The proof is based on an inductive argument considering the number of principals involved in the exchange. It relies on two intermediate results, which we prove first as Lemma 8.1 and Lemma 8.2.

**Lemma 8.1:** The algorithm will find and execute a sequence of actions which

1. is riskless,
2. excludes payment actions, and
3. results in the customer having the desired documents

if such a sequence exists.

For Lemma 8.1, the customer faces the further restriction that there are no payment actions. This restriction is enforced by the value of $\text{MoneyStatus}$ in the exchange between the two personas of C (like the one of Section 5.1). By setting the $\text{MoneyStatus}$ value to NOTHING instead of SENT, C will never advance money to another agent for this exchange.

**Proof:** (by induction of number of principals in the exchange, other than the final customer.)

**Base case:** The algorithm will find and execute a sequence of actions which is riskless, excludes payment actions, and results in the customer having the desired documents, if such a sequence exists, and involves 0 other principals.

If no other principals are involved (see Figure 10 (b)), it must be the case that the customer has the desired documents. Acquire.Set tests for ownership and records any document which is already held as being RECEIVED. If all documents are RECEIVED, check_for_next declares success without transferring documents or money. Therefore, the base case is established.

**Inductive hypothesis:** The algorithm will find and execute a sequence of actions which is riskless, excludes payment actions, and results in the cus-
Figure 10: A general conjunctive distributed commerce transaction.
customer having the desired documents, if such a sequence exists, and involves
$n$ other principals.

**Inductive step:** The algorithm will find and execute a sequence of actions
which is riskless, excludes payment actions, and results in the customer
having the desired documents, if such a sequence exists, and involves $n + 1$
other principals.

Figure 10 shows shows in part (a) a generic distributed commerce trans-
action. The ultimate customer, $C$, makes requests of one or more agents,
represented by Broker 1 through Broker $i$ in the figure. These brokers may
in turn contact other agents to assist them in obtaining the requested doc-
ments, though the specifics are unimportant from $C$’s point of view. There-
fore, these interactions are abstracted away, represented just as outwardly
branching triangles in the figure. The number of providers that $C$ contacts
causes a division of the scenarios into two classes: those where $C$ contacts
only one broker (in which case $i = 1$, and Figure 10(c) applies), and those
in which $C$ contacts several brokers ($i > 1$, Figure 10(a)).

**Case 1:** $i = 1$, $C$ contacts 1 broker

If there is only one document, **Acquire Set** sets up the records and calls
**check for next**. The fourth clause applies, and **Send Payment or Request**
is invoked. Since the customer’s banker persona has not authorized money
for this request, the **SR.client.MoneyStatus** is not SENT, and **Send Payment or Request**
defaults to $C$’s merely requesting the missing document. **Request Doc** causes
the contacted agent $B_1$ to try to acquire the document without risking any
money. Since this is a problem instance of at most size $n$, we know by the
inductive hypothesis that if there is a safe execution sequence, the algorithm
will find and execute it. Moreover, given the single source assumption, if
there is a riskless sequence for the whole transaction, $C$ must obtain the
document from broker $B_1$, so this broker must have been able to obtain it
risklessly, satisfying the antecedent of the inductive hypothesis.

Once the document is delivered to the source agent $B_1$, as $B_1$ is executing
**check for next** (triggered by the arrival of the final conjunct), $B_1$ will
discover it has all the conjuncts and will in turn call **Send Doc**. If the
document does not require payment, then it will be sent on, and $C$, the
ultimate customer, receives it. Therefore, the customer has managed to
obtain the document without transferring any money.
Case 2: $i > 1$, C contacts multiple brokers

Intuitively, if no money is to be transferred there is no risk of buying half a conjunction, and the individual conjuncts can be handled as in the previous $i = 1$ case. The ultimate customer is seeking several documents as a conjunction. Therefore, when \texttt{Acquire Set} is called, \texttt{check for next} will fall through to the fifth clause, and each document that has not been requested yet will be requested from agents B1 through Bi. As above, each of these individual conjuncts is a problem instance of size smaller than $n$, so if there is a way to transfer them to the appropriate B agent, the algorithm will find a way and execute it (according to the inductive hypothesis). As each of these conjuncts arrive at its individual B, \texttt{check for next} will result in the invocation of \texttt{Send Doc}, and if the document is free to C, it will be transferred to C, fulfilling the conjunction. Otherwise, there is no way to complete the exchange without a payment action, and since the antecedent of the Lemma is falsified, the whole Lemma is trivially satisfied. Therefore, if there exists a way to transfer documents to the customer without exchanging money, the algorithm will find and execute the sequence. This establishes our preliminary result.

\begin{lemma}
If a conjunction is risklessly obtainable, then at most one of conjuncts requires advance payment to obtain.
\end{lemma}

Equivalently, in a generalized distributed commerce transaction such as Figure 10(a), all or all but one of the B agents can acquire their documents as in Lemma 8.1, without transferring any money.

\begin{proof}
(by contradiction)

Assume there were a riskless way for a customer to obtain all the conjuncts of a conjunction, even though there are two (or more) documents which the broker agents must spend money to obtain (i.e., the brokers do not have the documents, and the sources that do have them demand payment from the broker). Then one of the following orders must occur:

- The customer must put up money for all the documents. Here the customer is risking money without a guarantee of getting the documents. One broker may provide the document and receive payment, while another broker does not. Although the money for the missed document is refunded, the customer still has spent half its money for a useless subset of the requested information.

\end{proof}
• The customer puts up nothing. In this case, the brokers are at risk. If they acquire the documents with their own funds, they run the risk of being stuck with the documents when the customer backs out. They have no recourse to return the documents, and have no mechanism to force the customer to buy the requested document.

• The customer pays for some subset of the documents. This approach combines the problems of both the previous approaches, while providing no benefit. The customer may still lose if the document(s) it paid for arrives but one of the other brokers fails to deliver. A broker that was not pre-paid for its document may acquire it, only to find the customer has disappeared.

Since any conjunction with at least two documents which require pre-payment in order to be obtained runs into one of the problems listed above, it cannot be a riskless exchange, contradicting the antecedent and validating Lemma 8.2. ■

**Theorem 8.1:** The algorithm will find and execute a sequence of actions which

1. is riskless, and

2. results in the customer having the desired documents

if such a sequence exists.

**Base case:** The algorithm will find and execute a sequence of actions which is riskless, and results in the customer having the desired documents if such a sequence exists and involves 0 other principals.

(Idential to base case for Lemma 8.1.) If no other principals are involved (see Figure 10 (b)), it must be the case that the customer has the desired documents. **Acquire** tests for ownership and records any document which is already held as being RECEIVED. If all documents are RECEIVED, **check_for_next** declares success without transferring documents or money. Therefore, the base case is established.

**Inductive hypothesis:** The algorithm will find and execute a sequence of actions which is riskless, and results in the customer having the desired documents if such a sequence exists and involves $n$ other principals.
**Inductive step:** The algorithm will find and execute a sequence of actions which is riskless, and results in the customer having the desired documents if such a sequence exists and involves \( n + 1 \) other principals.

As it did for the case of exchanges without money, the cardinality of \( i \) (the number of brokers contacted by \( C \)) makes a useful division into cases:

**Case 1:** \( i = 1 \), \( C \) contacts 1 broker

In this case, \( C \) has made its request to only one source, \( B_1 \), expecting \( B_1 \) to pull together all of the documents necessary to meet \( C \)'s information need. \( C \) invokes `check_for_next` as the final step of the `Acquire_Set`. The theorem requires that prepayment to trusted parties be permitted, therefore, the ultimate customer \( C \) must allocate money for this information purchase, (set `SR.client.TaskRecord.MoneyStatus` for the exchange with its “banker persona” to `SENT` before invoking `Acquire_Set`). Consequently, in the fourth clause of `check_for_next` \( C \) will agree to forward payment for the requested document to the intermediary \( T_1 \) that \( C \) shares with \( B_1 \). \( T_1 \) responds by signaling the provider \( B_1 \) with a `Notify_Payment` call. This signal enables \( B_1 \) to be sure that it has a guaranteed customer, allowing agent \( B_1 \) to make any necessary “out-of-pocket” payment to acquire the desired document. (N.B., We assume that broker agents have enough money to make the required payment. See Section 4.) Here we can apply our inductive hypothesis for purchases involving \( n \) or fewer principals. Since our final customer \( C \) is excluded from consideration, this task falls within the scope of our inductive hypothesis. Moreover, since \( C \) has advanced payment to \( T_1 \), the `MoneyStatus` for that exchange is `SENT`, yielding the required condition for the inductive hypothesis. Therefore, the algorithm will find and perform a safe execution sequence which brings the desired document to \( B_1 \); if such a sequence exists. (Given the single source assumption, such a riskless sequence must exist for \( B_1 \) to obtain the document, otherwise there can be no way for \( C \) to obtain the document. In that case the theorem will hold because the antecedent is falsified.) As soon as \( B_1 \) receives the document, the second clause of `check_for_next` is activated, and the document is sent (via `Send_Doc`) to the trusted intermediary \( T_1 \) shared with the ultimate customer \( C \). Since the intermediary has already received payment for this item, \( T_1 \) completes the exchange of this conjunct, calling the customer’s `Receive_Doc` method, and the broker’s `Receive_Payment` method.
The exchange is complete.

**Case 2: i > 1, C contacts multiple brokers**

If C instead divides the request into pieces for different brokers, the presence of more than one unacquired document immediately triggers the fifth clause in check for next, and all of the documents will be requested via the Request Doc function, without providing payment. In each case, the relevant principal B1 through Bi will perform an Acquire Set(Di) for the requested document Di. Lemma 8.1 shows that if the document may be obtained without exchanging money, the algorithm will find and execute a sequence which obtains it. Therefore, all of the conjuncts which can be obtained without pre-payment will find their ways to the relevant principals B1 through Bi, and when check for next recognizes that the documents have been obtained, it will call Send Doc, moving the documents to the trusted agent T1 through Ti, the corresponding principals B1 through Bi share with C. Trusted intermediaries T1 through Ti, recognizing that no money has been received for these documents, invoke Notify Doc on C, and the DocStatus for all of the documents at the trusted intermediaries is updated to SENT. Since these documents were obtained without the expenditure of money from the B agents, they have nothing to lose if the customer chooses to abort the transaction at this stage.

Combined with the knowledge of Lemma 8.2, we know that at most one conjunct will not have been obtained (be at Ti or at C) at this point. If "at most one" turns out to be zero, and all of the conjuncts are at the trusted intermediaries T1 through Ti, then check for next will determine that all SR.SourceTRs.DocStatus is SENT or RECEIVED, so the third clause is applicable. The final customer merely needs to iterate over the desired documents, sending the promised payment to each trusted intermediary, and receiving the document in exchange. Since the behavior of the intermediaries is known to be trustworthy, there is no risk that they will fail to provide the demanded documents. So in this case, the exchange may be completed without risk.

The remaining subcase is when there is exactly one conjunct that requires pre-payment. All the others have been requested and transferred to the trusted intermediaries between C and the acquiring broker agent. In this case, check for next finds that the fourth clause is activated, and C
advances payment for the missing conjunct to the trusted intermediary Ti shared with the the Bi agent that was unable to acquire its document without transferring money. The resulting outcome is exactly what it would be if the missing conjunct were the only one that C requested, (i.e., if \( i = 1 \)). If the missing conjunct can be obtained once payment is assured, then it will be obtained, and sold to \( C \), as shown in the proof of Case 1 above. \( C \) can then acquire the remaining conjuncts which are known to be at the trusted intermediaries. If the missing conjunct cannot be obtained, \( C \) receives a refund, and has lost nothing. Likewise, the agents Bi through Bi and any agents they have contacted have lost nothing either—their documents were acquired without transferring money.

For the case of all conjuncts obtainable without pre-payment and the case where exactly one requires pre-payment, it has been shown that the algorithm will find and execute a safe execution sequence if one exists. Furthermore, it has been shown that if two or more conjuncts require pre-payment, no such safe execution exists. Therefore, the inductive step holds, and our proof has been established.

9 Direct Trust

In the preceding discussion, we made the simplifying assumption that documents and money always are exchanged through trusted intermediaries because the principals did not trust each other. It may be the case, however, that one principal does trust another. This trust may manifest itself by one agent’s sending a document to another principal before receiving payment, expecting that the recipient will pay later or return the document if it is unusable. This willingness to trust may come from an established reputation held by one of the parties (e.g., if it is a government institution, or a well-known, established commercial venture), which would be damaged greatly by cheating a customer or supplier, or the belief may arise out of a history of interactions between the customer and supplier, where past exchanges have been conducted with integrity. The risk of losing all future business of a customer (and possibly other customers, if the word is spread), outweighs the gain that might be made on a dishonest completion of the current transaction. Finally, direct trust may be possible between parties because they
are operated by the same owner. Direct trust need not be symmetric. A customer may trust a merchant without the merchant trusting the customer. Direct trust manifests itself in two ways:

1. If a provider trusts a customer, the provider will send a document directly to the customer even before receiving payment, trusting that the customer will subsequently pay for or return the document.

2. If a customer trusts a provider, the customer may advance payment to the provider, trusting that the money will be returned if the document cannot be obtained.

At first glance, the latter point seems trivial. The difference between a direct payment and the guaranteed notification from a trusted intermediary is minor, and does not affect the feasibility of transactions under the assumption that document prices are negligible compared to agents’ resources. If this assumption is violated, however, direct trust allows some of the customer’s money to be spent by the broker to obtain the desired document. In contrast, if the broker receives only a promise of payment from a trusted intermediary, it will not have sufficient funds to purchase the document and the exchange would fail. Further exploration of this issue is grounds for future work.

### 9.1 Changes to Algorithm

When there is direct trust between principals, the changes required to the algorithm are relatively minor. The difference in behavior, however, may be significant. In each instance where an item was formerly sent to the trusted intermediary, a check is made to see if the recipient is trusted directly. If so, the intermediary is bypassed. The only difference between the case where the receiving principal is trusted and the one where it is not is the target object of the `ReceiveDoc` call. In either case, the goods are sent as far “downstream” (as close to the customer) as possible without violating trust conditions.

Another addition is required to ensure the fulfillment of obligations undertaken as a result of being trusted. If a customer has received a document from a source that trusts it, it must pay for the document if the rest of
the exchange is completed, or return the document if it is aborted. For a successful completion, (detected in the second clause of check_for_next if the agent is the ultimate customer or in ReceivePayment if acting as a broker), the MoneyStatus of each document in the SourceTRs set is examined to be sure that payment has been sent, otherwise it is sent immediately to the source. For each unsuccessful completion, (e.g., unknown_document, not_in_time, and multiple calls to Request_Payment within one conjunction) each document received on trust must be returned to the source. In the event of one of these failures, any payment that the customer advanced payment to a broker or source must be returned as well. These refunds occur in the same places of the algorithm that document returns do.

9.2 Successful Exchange in Example 2 with Direct Trust

Certain exchanges which are infeasible without direct trust become feasible once it is added. The second example discussed in Section 6 (Figure 8) was previously infeasible. However, if S1 trusts B1, the exchange becomes feasible, as we now demonstrate. The initial steps are the same, up until S1 has discovered that it has D1, all of the documents required to fulfill B1's request (clause #2 of check_for_next). SendDoc is still called, but because S1 trusts B1, the document is sent directly to B1 rather than to T3. Since the exchange is ongoing (B1 has not yet received payment for D1), B1 is not obligated to pay S1 for D1 yet, but will pay for it later when and if C buys the document. When B1 receives the document, it updates the state of the transaction and calls the dispatch function check_for_next on the parent of this exchange, namely the exchange between B1 and C. Since B1 has received D1, it finds that the second clause is activated. Once again, Send_Doc is invoked, but since there is no trust relationship between B1 and C, the document is sent via trusted intermediary T1.

When T1 executes Receive_Doc, it recognizes that no payment for this document has yet been received, so it invokes the Notify_Doc method on C. The customer consequently updates its status of D1 to SENT, so that when it invokes check_for_next, only D2 is missing. Therefore, the fourth clause is activated which permits pre-payment for one conjunct. Send_Payment_or_Request is invoked with SR_1 as the parameter, and since
C has received the money for this document from its client, the banker "persona" of C, so C is able to advance money to T2 in order to obtain D2. C does this using the Receive_Payment method on agent T2. T2 checks to see if this payment enables it to complete an exchange, but finds that it has not yet received D2. Therefore, it stores the payment and triggers a Notify_Payment on B2. Rather than creating a new SR, B2 recognizes that this payment is for a document that was previously requested, B2 updates its SR, and calls check_for_next. The routine check_for_next initiates a similar series of steps which allowed C to obtain the document in Example 1 of Section 5. This sequence results in C obtaining D2 and ends with a call to check_for_next. With D2 in hand and D1 at T1, C’s invocation of check_for_next dictates sending payment to T1 in order to obtain D1.

T1, executing the Receive_Payment method, completes the exchange, sending D1 to C, and sending the payment to B1. During B1’s execution of Receive_Payment, it checks to see if it owes money for any of the source documents (there is some sourceTR for which Money_Status is not SENT or RECEIVED). Those documents were received on trust, and now that the broker has been paid for them, the obligation to re-pay the sources that provided them is in full force. B invokes the Receive_Payment method on S1 in order to fulfill this obligation. At this point, B1 and S1 both call check_for_next to see if there is anything further to do with respect to this transaction. Finding nothing applicable in check_for_next, they conclude correctly that the exchange is complete.

9.3 Soundness

Direct trust weakens the restrictions that agents place upon the exchanges. They no longer insist that they receive complete protection against deviant behavior from all the principals in the exchange. In particular, there are certain agents who are directly trusted, who are permitted greater latitude than other untrusted agents. A source will send a document to a directly trusted principal, even before payment has been guaranteed, believing that
the recipient will eventually pay for the document or return it in good faith. A customer who directly trusts a source is willing to send payment before receiving the goods.

**Theorem 9.1:** Any sequence of actions produced by the algorithm modified for direct trust is riskless with respect to the direct trust safety conditions, assuming trusted intermediaries follow the algorithm.

**Proof:**

We show in turn that each of the four unsafe conditions cannot arise in a sequence of actions generated by the algorithm.

1. *Whenever a customer sends money, it always receives the document or a refund:* In the unmodified version of the algorithm, a customer only gives money to trusted intermediaries, who provide either the document or a refund. In the version of the algorithm modified to permit direct trust, the customer will occasionally send payment directly to a trusted provider (in either `Send_Payment_or_Request` or `Send_All_Payments_or_Request`, the customer invokes `Receive_Payment` on the provider). If the provider follows the algorithm and is able to acquire the document, the second clause of `check_for_next` will hold, and the provider will invoke its own `Send_Doc` method, resulting in (since the `MoneyStatus` is RECEIVED) the document being sent directly to the customer. If the provider fails to obtain the desired document, the money is returned to the customer (in `expire` which may be called by `unknown_document` or `Request_Payment` or a deadline failure). Therefore, any sequence the algorithm generates will ensure that whenever a customer spends money without having it refunded, the customer will get the requested document.

2. *Whenever a provider sends a document, it always receives payment or the document is returned:* Similar to the previous case. If the document is sent via a trusted intermediary, the document is available to the customer only when the payment is also sent to the provider. If the document is sent directly to a trusted customer, the provider receives payment when the customer obtains all of the desired documents and is paid for them (in clause two of `check_for_next` if the customer is the ultimate customer, or in `Receive_Payment` if the customer is a
broker). If the customer is unable to get the rest of the documents in the conjunction, the document sent on trust is returned (in \texttt{expire} called either by \texttt{unknown\_document} or \texttt{Request\_Payment} or clause one of \texttt{check\_for\_next} or a failure from another agent). Therefore, any sequence the algorithm generates will ensure that whenever a provider sends a document without having it returned, the provider will get receive payment, or a trusted agent did not follow the algorithm.

3. A customer never buys half a conjunction: Direct trust does not induce a customer to buy more documents. The same conservative policy of the unmodified algorithm is followed in the modified version; the only difference is to whom payment is sent.

4. A broker never buys a document without being able to re-sell it: Here again, direct trust does not impact the decisions of agents with respect to purchasing documents. Therefore, the conservative policy followed in the original algorithm works for this case as well.

Therefore, if none of the four unsafe cases may arise from the sequences suggested by the algorithm, each agent finds the sequence safe, so it is riskless for the population as a whole. Consequently, any sequence suggested by the algorithm is riskless. ■

9.4 Completeness

Just as the algorithm without direct trust was proven complete with respect to the unsafe sequences described in Section 7, the algorithm with direct trust must be shown complete with respect to the unsafe states described above. For the algorithm to be complete, it must be the case that if there is some riskless sequence which will result in the customer obtaining the desired documents, then the algorithm will find such a sequence. The proof of completeness for the algorithm with direct trust mirrors the earlier version in Section 8. For Lemma 8.1, we substitute:

\textbf{Lemma 9.1:} The algorithm, as modified to permit direct trust, will find and execute a sequence of actions which

1. is riskless (with respect to the direct trust conditions),
2. excludes payment actions, and
3. results in the customer having the desired documents

if such a sequence exists.

Proof: (by induction on number of principals in the exchange, other than the final customer.)

Base case: Identical to that for Lemma 8.1.

Inductive hypothesis: The direct trust algorithm will find and execute a sequence of actions which is riskless (with respect to direct trust), excludes payment actions, and results in the customer having the desired documents, if such a sequence exists, and involves \( n \) other principals.

Inductive hypothesis: The direct trust algorithm will find and execute a sequence of actions which is riskless (with respect to direct trust), excludes payment actions, and results in the customer having the desired documents, if such a sequence exists, and involves \( n + 1 \) other principals.

The proof of this step is identical to that for Lemma 8.1, with the exception that the document will be sent to the final customer \( C \) under two conditions:

1. The document is free to \( C \) (identical to Lemma 8.1), or
2. The relevant broker \( B \) directly trusts \( C \) (unique to direct trust).

Therefore, if a provider trusts a customer, the document may be sent on without having payment guaranteed at a trusted intermediary. This modification ensures that if direct trust enables a document to be sent without payment, the algorithm finds the sequence, fulfilling the inductive step. ■

Lemma 8.2 continues to hold with the conditions of direct trust, reinforcing the inability to obtain conjunctions where multiple conjuncts require advanced payment. That is, an agent who gives money to a trusted principals who are to obtain the conjuncts of a conjunction is putting that money at risk. Even if all the brokers behave in a trustworthy manner, the customer still risks buying half a conjunction if some conjuncts are unavailable.

Theorem 9.2: The algorithm will find and execute a sequence of actions which

1. is riskless (with respect to direct trust), and
2. results in the customer having the desired documents

if such a sequence exists.

Proof: (by comparison to Theorem 8.1)

Using Lemma 9.1 and Lemma 8.2, the direct trust version of Theorem 8.1, follows in the same manner as in Section 8. Direct trust requires only minor changes to the proof. In the proof of the inductive step, Broker Bj may choose to send the documents to final customer C rather than the trusted intermediary Tj if Bj directly trusts C. C can then acquire the full set of documents by sending payment to trusted intermediaries for any documents that were not sent directly. C will pay for the documents received due to direct trust at the end of the process (in check_for_next, clause 2). If the transaction is aborted instead, Bj trusts C to return the document in good faith.

The rest of the completeness proof of Theorem 8.1 holds without modification, establishing the completeness of the algorithm modified for direct trust. ■

10 Timing and deadlines

In the previous examples, when a customer sends payment to a trusted intermediary, the customer cannot control when it gets its money back. For this to be a realistic solution, there needs to be a mechanism by which the customer can establish a deadline for receiving the ordered goods, with the ability to get a timely refund if the deadline is not met. In this section, we introduce such a mechanism which allows agents to establish deadlines by which exchanges must be completed. This change requires additions to the TaskRecords, as well as the algorithm itself. We will start by describing the model of time that we use, then show an example of deadlines in use. Next we will describe the changes to the data structures and algorithm. Finally, we will return to the formal properties of the algorithm, showing its soundness and completeness under this formalization.

We will model time using Lamport’s representation of physical clocks for distributed systems [4]. Each clock value is a number, and larger numbers represent later times than smaller ones. We ignore the problem of clock synchronization, assuming that all of the agents’ clocks show the same value at
one instant in time. Agents also have knowledge of how long delivery may
take between any two agents, and delivery times are symmetric between
pairs of agents, which is represented by a function unique to each agent
called $\text{DELIVERYTIMETO}(x)$, which returns the upper bound for the amount
of time that a delivery sent to neighboring agent $x$ will take, including any
operations by trusted intermediaries in the exchange. Finally, in this ideal-
ization, the time requirement of local computations is negligible compared
to communication time.

10.1 Example using Time

In this section, we will repeat the first example under varying time con-
ditions, showing one example where the transaction is still feasible, and a
second where it cannot be completed in time. Intuitively, the changes are
minor. A deadline is added to each document request. In order for the
request to be satisfied, the client must receive the document at or before the
deadline. Agents may use their knowledge of transfer times (recorded in the
$\text{DELIVERYTIMETO}$ function) to check whether an exchange will finish in time
before they commit to undertake a costly action for that exchange. Like-
wise, the trusted intermediaries always check that the exchange will meet
the customer's deadline before sending the goods to the customer. If it
will not, the pieces are returned to their respective owners. Since the value
in $\text{DELIVERYTIMETO}$ is an upper bound, however, in some cases an agent
should take a riskless alternative (such as requesting documents rather than
forwarding payment) in the event that the operation may be completed be-
fore the pessimistic upper bound, allowing the rest of the transaction to
complete.

For the first version of Example 1 with deadlines (see Figure 11(a)),
assume that all of the delivery times are at most 5 seconds between an agent
and a trusted intermediary, and at most 10 seconds between two principals.
At time 30, $C$ places an order for document $D$, and wants it before time
75. $C$ creates the same TR as it did in Section 5, but this time adds the
deadline information, that $D$ must be received at $C$ no later than time 75.
This request and the money are sent to $T_1$, arriving at or before time 35.
$T_1$ alerts $B$ using $\text{Notify}_\text{Payment}$, and $B$ learns of $C$'s request at or before
time 40. The request is still marked with $C$'s deadline of time 75.
Figure 11: Examples of temporally constrained exchanges.
B realizes that in order to meet the deadline, B must start sending D to C no later than time 65, since it may take as much as 5 seconds to get to T1 and another 5 to get from T1 to C. But B does not have D, so it must ask S to provide it. Since we assume that local processing time is negligible, B can receive the document from S as late as time 65 and still guarantee that C will have it at time 75, before the deadline is reached. Therefore, when B dictates a deadline to S, it is 65, the last possible instant that B must have the document in order to fill its client's order.

B's order reaches S (via T2) at or before time 50. S has no knowledge that D is eventually destined for C, but still knows (via a field in the TR) that the ultimate deadline for this exchange is 75. S recognizes that it does have D and can dispatch it right away. S sends D to T2, who receives it no later than time 55, and sends it on to the broker, where it arrives no later than time 60. B immediately turns the document around, sending it to T1, where it arrives at or before time 65. T1 already has C's money, so is able to complete the exchange immediately, sending D to C, where it arrives at or before time 70, with time to spare before the deadline is reached.

In the second variant (Figure 11(b)), C demands having D at time 65, a full 5 seconds before the agents were able to guarantee the arrival of the document in the previous variant. As we will see, this second variant is infeasible. The initial steps proceed as in the first variant. C sends a requesting TaskRecord with a deadline of 65, which arrives at B at time 40. (We will assume the worst case here, with each transfer taking its upper bound.)

B sends its request to S with a deadline of 55, noting that the ultimate deadline for this exchange is 65. If S is able to meet the earlier deadline of 55, B will be guaranteed to be able to send the document back to C in order to meet its deadline of 65. But S recognizes, when it receives the request at time 50, that even if sends the document immediately to T2, it may still not arrive at B before time 60, which is after the deadline of 55 that B has imposed. On the other hand, if the document moves very quickly through T2, B, and T1, it could arrive well before the ultimate deadline of 65. Therefore, S knowing that this exchange may fail, has nothing to lose by sending document D to the trusted T2, the trusted intermediary shared with B. If for some reason the connection between S and B goes very quickly,
then perhaps B can still fulfill C’s query.

When D does not arrive at T2 until time 55, however, T2 recognizes that the exchange cannot be guaranteed to complete in time. Even if T2 sent the document to B and it arrived in one second at time 56, B might still not be able to get it to C in time, since B’s \( \text{DELIVERYTIME}_C \) is 10, which would place the document at C at 66, after C’s deadline. Therefore, T2 returns B’s money and S’s document, and calls \texttt{expire} on the adjacent parties to abort the other exchanges in progress. The expire message passes down to B and C, returning the money that C had sent to T1. C is faced with the choice of giving up its request for D, attempting to locate an alternative source, or trying again with B, but offering a longer deadline.

### 10.2 Changes to Data Structure and Algorithm

The changes to the data structures are minimal. A new field entitled “Deadline” is added to the TaskRecord. The documents requested in the TR must be received at the consumer at or before this clock value. A second field called “Ultimate Deadline” is also added to the TaskRecord. It contains the ultimate customer’s deadline for this exchange.

The presence of two deadlines (one for the direct customer, one for the ultimate customer) seems redundant, but in fact they provide different information. The earlier deadline guarantees payment, if it is met. If the later deadline has passed, then there is no chance the exchange will succeed, and an agent need not bother taking any further action, other than invoking \texttt{expire}. If the earlier deadline has expired, but the later one has not, then the agent should take costless actions which may allow the exchange to proceed. In this “gray area”, the exchange may succeed if transit times are less than their worst case \( \text{DELIVERYTIME}_C \) values, but it is not guaranteed.

Before any “expensive” action is taken (such as acquiring a document on behalf of another or sending a document to its requester), a check is made to ensure that the customer will receive the document in time. If not, the expensive action is not taken, but a riskless action is substituted, such as making a request. The timing check is implemented in the method \texttt{checkTime}. Before making a payment, a provider invokes the \texttt{checkTime} method to determine the maximum amount of time required for it to acquire all of the source documents (which are currently at the neighboring
agents). This calculation allows for multiple documents to be sent at the
same time, so the total time for receiving all of the documents is just the
amount of time until the longest individual document arrives, not the total
of all the individual times. It also takes into account the document’s point
of origination, knowing whether the document is at the trusted intermediary
or must travel “two hops” from the source. The deadline that the principal
must meet was established by its customer as the time the customer must
have the documents. So check_time considers not only the time that this
agent will take to receive the last needed document, but also the time until
the agent’s client get the resulting document from this agent.

If the deadline will be met, check_time returns true, allowing processing
in the calling function to continue, otherwise it returns false and no further
costly actions are taken. However, in the event that the riskless alternatives
complete successfully and speedily, it may be possible to continue, even
though it was not possible to offer a total guarantee in advance. Calls to
check_time are invoked in two places: in Send_Payment_or_Request and
send_all_Payments_or_Request.

A second aspect of the temporal enforcement is the first clause of check_for_next.
Whenever any agent executes check_for_next, they first ensure that the ul-
timate deadline for this exchange has not yet passed. If the ultimate dead-
line is passed, then there is nothing that can be done for this exchange. The
agent that discovers the problem should return all goods or money received
on trust for this exchange, and let its sources and client know that they
should also end their involvement in it.

The trusted intermediaries also have a central role in enforcing deadlines.
An exchange which might not be completed by the deadline will not be
executed by the trusted intermediary, even if both components are present.
The modification to Receive_Doc is quite simple, just checking that the
document will arrive at the customer before sending either it or the payment.

The trusted intermediaries periodically check through the set of ex-
changes they are mediating in order to determine if any exchanges have
expired, and the received goods should be returned to the principal owners.
In addition to checks that are triggered by the passage of time, such house-
keeping may be triggered by an expire method that is called by one of the
principals involved in an exchange mediated by the trusted intermediary.
A further modification is required for the temporal version of the algorithm. Whenever an agent agrees to act as a broker, it must specify a deadline to its source. Moreover, that deadline must give the broker enough time to send the resulting document on to the ultimate customer before the customer’s deadline is reached. It sets the deadline by taking its customer’s deadline, and subtracting the sum of the transit times from the provider to the agent itself and the agent to the customer. For the ultimate deadline, the broker agent merely passes on whatever is in its client’s TR. Obviously, this value cannot be earlier than the direct deadline for the pairwise exchange.

10.3 Soundness

The introduction of time into the system imposes several new requirements that must be fulfilled to have a sound (satisfactory) solution. The main condition is that the customer receive the document before the expiration of a deadline that it establishes.

Note that the whole exchange need not be completed by the deadline. In particular, the source may not receive its money from the trusted intermediary before the deadline. As long as the customer will receive the document in time, the exchange will proceed. The source may be required to wait the differential in delivery times for items originating from the trusted intermediary going to the source and client before receiving its money. If the customer is going to receive a refund, it is not critical that the refund arrive before the deadline, just that it be sent near that time.

Here, we show the soundness conditions (similar to those established in Section 7), and prove that the proposed algorithm satisfies these conditions.

**Theorem 10.1:** Any sequence of actions produced by the temporal version of the algorithm is riskless with respect to the temporal safety conditions:

1. Whenever a customer sends money, it always receives the document before the specified deadline, or it receives a full refund.
2. Whenever a provider sends a document, it always receives payment, or the document is returned.
3. A customer never buys half a conjunction.
4. A broker never buys a document without being able to re-sell it.

Proof:

We show in turn that each of the four unsafe conditions cannot arise in a sequence of actions generated by the algorithm.

1. Whenever a customer sends money, it always receives the document before the specified deadline, or it receives a full refund:

A customer will only send money to a trusted party. A trusted intermediary will return the money unless the provider sends the document (invokes the \texttt{ReceiveDoc} method on the trusted intermediary) and the trusted intermediary is certain that the document will reach the customer in time (\texttt{checkTime} returns true). If the money is sent directly to a trusted principal, it will be returned in the event of a deadline failure or any other case where the document cannot be provided. Each failure (unknown document, deadline expiration, multiple payment requests) results in the invocation of the method \texttt{expire}, on the agent that first discovers the failure. The expiration messages are passed back to clients and forward to sources until all agents involved in the transaction learn that it cannot be completed. At each step along the way, the principals return documents received on trust to their providing sources, and return money received from trusted customers. Therefore, if the exchange does not succeed, the customer will receive a refund either from the trusted intermediary or from the trusted principal (executing \texttt{expire()}). On the other hand, if the exchange succeeds, the customer will have the desired document.

2. Whenever a provider sends a document, it always receives payment, or the document is returned: This condition is unaffected by the addition of temporal issues, and the mechanisms from the direct trust version of the algorithm continue to provide sufficient protection in the temporal version as well.

3. A customer never buys half a conjunction: This condition is unaffected by the addition of temporal issues, and the mechanisms from the direct trust version of the algorithm continue to provide sufficient protection in the temporal version as well.
4. A broker never buys a document without being able to re-sell it:

This condition is dependent on two aspects. First, the atemporal part that the customer may back out of a purchase. This threat is removed as it was before, by ensuring that the customer has provided money to the trusted intermediary, \textit{MoneyStatus} is \textit{SENT}, before it attempts to acquire a document.

The second part of this soundness condition is that the broker must be able to re-sell the document before the customer's specified deadline. To prevent this risk, the broker undertakes a second test, which permits money to be spent for acquiring a document for re-sale only if \textit{check_time} guarantees that the rest of the needed documents can be assembled and sent to the customer in time. These \textit{check_time} tests occur in both entry points to \textit{send\_Payment}, namely \textit{send\_All\_Payments\_or\_Request} and \textit{Send\_Payment\_or\_Request}. If the broker cannot guarantee that all the customer's request can be completed by its stated deadline, the broker will not spend money in its execution.

Therefore, if none of the four unsafe cases may arise from the sequences suggested by the algorithm, each agent finds the sequence safe, so it is riskless for the population as a whole. Consequently, any sequence suggested by the algorithm is riskless. \hfill \blacksquare

10.4 Completeness

Having described the unsafe states for a distributed exchange with deadlines, here we prove that if there is an exchange which is riskless and does not result in any agent's suffering a violation of one of the soundness conditions, then the algorithm will find that exchange or an equivalent one. There are two further assumptions which we view as reasonable requirements of the temporal algorithm.

\textbf{Assumption 1:} A direct communication path is no slower than an indirect communication path. For instance, if $C$ wants to make an information request from its neighbor $B_1$, there is no way to do so which is faster than invoking $B_1$'s \textit{Request\_Document} method.

\textbf{Assumption 2:} An agent may not take steps to fulfill an information request before being notified of that request. For instance, an information
source may not send a document to a broker in advance of having it requested by that broker.

As in the previous completeness proofs, we proceed with first showing that the algorithm is complete for sequences which do not require payment actions.

**Lemma 10.1:** The algorithm, with modifications for direct trust and deadlines, will find and execute a sequence of actions which

1. is riskless (with respect to the temporal soundess conditions),
2. excludes payment actions, and
3. results in the customer having the desired documents as quickly as possible

if such a sequence exists.

**Proof:** (by induction on number of principals in the exchange, other than the final customer.)

**Base case:** Identical to that for Lemma 8.1, with the added consideration of time. Since there are no messages, money, or documents exchanged, no time is consumed, and the agent gets the documents as quickly as possible, with no time spent.

**Inductive hypothesis:** The temporal algorithm will find and execute a sequence of actions which is riskless (with respect to deadlines and direct trust), excludes payment actions, and results in the customer having the desired documents as quickly as possible, if such a sequence exists, and involves at most \( n \) other principals.

**Inductive hypothesis:** The temporal algorithm will find and execute a sequence of actions which is riskless (with respect to deadlines and direct trust), excludes payment actions, and results in the customer having the desired documents as quickly as possible, if such a sequence exists, and involves at most \( n + 1 \) other principals.

Here again, the proof parallels that for Lemma 8.1, as extended in Section 9 for direct trust between principals. In order to complete the proof of the Lemma, we need to show that the algorithm respects deadlines, and delivers the documents as quickly as possible.
First, in the restricted case of the Lemma, we are considering only those execution sequences which do not require payment actions. Therefore, the second soundness condition (that a source always receives the necessary payment) is trivially satisfied, since the source was not expecting any payment. Likewise, the customer does not have to pay for any document, whether it is delivered before or after the deadline, making the first soundness condition trivial. Therefore, deadlines do not change the requirements of the algorithm.

Second, in the restricted case of the Lemma, documents are always delivered as quickly as possible. When the customer C attempts to acquire the documents for a problem involving several agents, either clause four or five of \texttt{check for next} applies. In either case, the customer immediately calls \texttt{Request Doc} from the appropriate providers, call them B₁ to Bₙ, (and by assumption this direct approach is the quickest way to contact an agent). The resulting request involves at most n agents (since the ultimate customer is excluded), and falls within the scope of the inductive hypothesis. Therefore, the algorithm generates an execution sequence that moves the desired documents to B₁ through Bₙ, and gets them there as quickly as possible, if any sequence will get them there. Once the documents arrive at the Bᵢ agents (using the \texttt{Receive Doc} method), \texttt{Receive Doc} calls \texttt{check for next}, triggering execution of the first case, whereupon the Bᵢ agent will invoke its own \texttt{Send Doc} method. Since no payment is required (by restriction of the Lemma), no trusted intermediary is necessary, and Bᵢ sends the document directly to the ultimate customer, the fastest possible transport available. So, C used the fastest possible approach to contact the B agents, they (by the inductive hypothesis) acquired the documents as quickly as possible and sent them directly to C, the fastest possible delivery. Therefore, C got the documents as quickly as possible, subject to the assumptions that agents cannot take actions before being notified of the request, and agents may not contact providers outside their sphere of knowledge. ■

The result of Lemma 8.2 on having at most one conjunct that cannot be acquired without payment is still in force in even when conditions of direct trust and deadlines are added. It will again be used to prove the final completeness result with deadlines.

\textbf{Theorem 10.2:} The algorithm will find and execute a sequence of actions
which

1. is riskless (with respect to the temporal soundness conditions), and

2. results in the customer having the desired documents as quickly as possible

if such a sequence exists.

Proof: (by induction on number of principals in the exchange, other than the final customer.)

Base case: The algorithm will find and execute a sequence of actions which is riskless, and results in the customer having the desired documents as quickly as possible, if such a sequence exists and involves 0 other principals.

(Idential to base case for Lemma 10.1.)

Inductive hypothesis: The algorithm will find and execute a sequence of actions which is riskless, and results in the customer having the desired documents as quickly as possible if such a sequence exists and involves at most \( n \) other principals.

Inductive step: The algorithm will find and execute a sequence of actions which is riskless, and results in the customer having the desired documents as quickly as possible if such a sequence exists and involves at most \( n + 1 \) other principals.

As it did for the earlier proofs, the cardinality of \( i \) (the number of brokers contacted by C) makes a useful division into cases:

Case 1: \( i = 1 \), C contacts 1 broker

After creating the SetRecord in \texttt{Acquire\_Set}, C invokes its own \texttt{check\_for\_next}. In this case, C makes its request to only one source, B1, expecting B1 to pull together all of the documents necessary to meet C’s information need. Therefore, the fourth clause of \texttt{check\_for\_next} is applicable, invoking \texttt{send\_Payment\_or\_Request}. This routine searches through the SetRecord, determining which conjunct is still missing. It requests that document directly. If B1 is able to obtain the document without C’s guarantee of payment, then this direct contact may be swifter than waiting for the trusted intermediary to notify B1, possibly permitting the extra time that makes the difference between the provider making the deadline instead of missing it.
However, B1 may need the promise of payment before it can acquire the necessary documents. Therefore, C still attempts to see if it can guarantee the payment by sending money to the shared trusted intermediary. C can only do that if two conditions are met:

1. C’s customer (in this case, the banker persona) has advanced payment, and

2. even if transit times take their worst case upper bounds, C can still receive its goods in time and provide them to its customer.

In this instance, both of these conditions are met, since C’s banker persona has allocated the money, and the transit time between C and its banker persona is zero. Therefore, if B1 can meet the deadline that C has established, there is no extra requirement for sending to C’s banker persona. If either of these conditions were violated, C’s request (not backed up by payment) may still result in the satisfaction of the query. However, it would not be riskless. In the worst cases, the documents would take their maximum transit time, and C could not re-sell its document in time, resulting in a violation of the fourth soundness condition. Therefore, C does not put any of its resources at risk, but rather takes the costless action of requesting the document rather than sending payment for it.

C sends the payment either to B1 if B1 is trusted by C, or to their shared trusted intermediary otherwise. This combination of actions ensures that B1 is contacted as quickly as possible by C. A direct request is made immediately, allowing B1 to obtain the document if no payment is required. If payment is required, it is sent to B1 as quickly as possible: directly from C if B1 is trusted by C, or via a trusted intermediary otherwise.

Once B1 has been contacted with the request, B1 continues the algorithm, and by the inductive hypothesis, we know that if a riskless sequence exists, B1 will find and execute it, obtaining the documents as quickly as possible. B1’s performance is dependent, of course, on whether C merely requested the documents or made a guarantee to pay by sending money to B1 or their shared trusted intermediary. In the former case, B1 will not risk any of its own resources, while in the latter, C will spend from its own pocket if it is certain that the document can be acquired and sent to C in time.
The final stage in the process is to return the documents to C as quickly as possible. This step begins when B1 recognizes that it has all of the desired documents (case two of check_for_next, which was invoked by receive_Doc). B1 determines that it was not the ultimate customer, and calls upon its send_Doc method to deliver the document to C. At this stage, the document will be sent, even if it is not guaranteed to arrive before the deadline, unless, of course, the ultimate deadline has passed, and case one of check_for_next applies.

The provider of the document is still safe when sending the document. The document will be sent to the requesting principal only if that principal is trusted by the provider. Otherwise, the document is sent to the shared trusted intermediary. The document must be sent, because even though there is no guarantee that it will arrive in time, the delivery times are worst case values. If the delivery goes smoothly, the document might still arrive in time. If the recipient is trusted, then the provider can expect to have the document returned in good faith if it did not arrive in time, or receive payment if it did. If the eventual recipient is an untrusted principal, then the document is sent to a trusted intermediary instead, using the Receive_Document method on the intermediary.

The intermediary performs the ultimate evaluation in Receive_Document. Unless the document is guaranteed to get to the requesting principal in time, given the worst case value of DELIVERY_TIME, the document is returned, any payment from the customer is returned, and the exchange is cancelled. If there is still time, the intermediary will complete the exchange if payment has been made. If payment has not yet been made, the trusted intermediary will invoke the Notify_Doc method on the customer, showing that the exchange can be completed upon presentation of payment. This method in turn calls the customer's check_for_next, and if all of the conjuncts are now available (clause 3), the customer will try to send payment. However, in Send_All_Payments_or_Request, the customer ensures that the exchange can be completed using the check_time call.

Therefore, in the case where the customer contacts only one provider, the algorithm will generate a riskless execution sequence for any problem has one. Moreover, that execution sequence will result in the customer obtaining the desired documents as quickly as possible. If an exchange is
not guaranteed to complete in time, the principals will continue to pursue it, but not commit any resources. The trusted intermediary acts as the final gatekeeper, delivering the goods only if payment is in hand and the document is guaranteed to arrive before the customer's deadline.

**Case 2:** $i > 1$, C contacts multiple brokers

As in the atemporal version, when the customer makes a conjunctive request, there is the additional restriction that the customer cannot advance payment to more than one provider without risking buying half a conjunction. The temporal version handles the risk in the same way as the atemporal version, reducing the customer's attempt to acquire the desired documents to a simple request, with no payment guaranteed. The addition of deadlines does not make any significant difference for the conjunction. Each branch of the conjunction can be handled separately, as in the previous case. When all but the last conjunct arrive, the customer may send payment for the final one, establishing a deadline that will allow the delivery of the missing conjunct, along with payment for and delivery of the other conjuncts.

To review, we have proven, using an inductive argument, that the customer will use the most expedient method to notify the provider of the request and the status of payment for that request. Money will be paid (sent to a trusted provider or trusted intermediary) whenever it may be done so without risking violation of the soundness conditions. The provider will acquire the documents as quickly as possible, and dispatch them to the customer as quickly as possible. Moreover, the delivery always involves a trusted agent (intermediary or principal) who moderates the exchange, protecting the document from disclosure without payment, or disclosure after the deadline. ■

### 11 Early failure notifications

In this section, we consider several modifications which improve the efficiency of the process, explicitly alerting a customer that the exchange will fail as soon as the failure is discovered, rather than waiting for the customer-imposed deadline to expire.
11.1 Unknown Document

The first optimization is for a source to notify a customer immediately if it has no way of obtaining a document. This addition requires only a slight change to the Acquire_Set method, which is responsible for finding the appropriate source for a document. If the agent knows of no appropriate source, it can inform its client immediately, which saves the client from having to wait for expiration of the deadline. The agent accomplishes this by returning any money or documents given to it in trust (using the expire method, and invoking the application specific unknown_document method on the client. At this point, the client can attempt to locate another source, or decide to submit a modified query or document request.

11.2 Not in time

A second optimization which allows the customer to detect a failure before the expiration deadline is to allow the client or the trusted intermediary to announce that it has stopped trying, because the deadline could not be guaranteed. If transit times for documents are long, this savings can be substantial. The changes required are minimal. When a trusted intermediary discovers that it cannot ensure the exchange will complete on time, it invokes the expire method, thereby returns the money to the client and the document to the provider. The expiration calls percolate up and down through the pair-wise exchanges until at last, all of the agents involved in the global exchange have been notified, and the ultimate customer alerts the application to the failure of this transaction.

11.3 Request Payment

The third, and most complicated, optimization allows the customer to determine when there is a series of dependencies in a conjunctive request which prevents its fulfillment, due to the risk of buying half a conjunction. This condition arose in the conjunctive example of Section 6. Customer C was unwilling to forward payment to either broker B1 or B2, but the brokers were also unable to get the documents from sources S1 and S2 without guaranteed payment from C. If both brokers informed C that they were awaiting payment, C would recognize that there was a deadlock that could not be
resolved in a riskless fashion. C could then attempt to find an alternative means of acquiring the document, or choose to take the risk by forwarding the money to one or both of the brokers. The optimization described here provides a mechanism for the brokers to inform C that they need a payment guarantee before they can obtain the documents.

Clauses three and four of check_for_next call for the agent (in this case broker B1 or B2) to check the value of client.MoneyStatus before making pre-payment to acquire documents for its client. If its customer (in this case C) has guaranteed payment by giving money to the trusted intermediary (MoneyStatus = SENT), then the agent may safely continue its plan to acquire the documents.

Consider a broker that has received a request for a document. If the customer has not sent payment, and the broker is unable to obtain the requested document without sending money, then the broker should invoke the Request_Payment method on the customer. Since a customer following the algorithm always proactively sends pre-payment whenever it is safe, the fact that this customer has not sent payment indicates that it cannot without risking an unacceptable end to the transaction (buying half a conjunction). The brokers realize that this request will not actually make the customer any more likely to pay, but that their actions improve the efficiency of the system as a whole. These payment requests must be added to the Send_Payment_or_Request and Send_All_Payments_or_Request methods. The Request_Payment method that the client implements is designed to see if any other agent working on part of this conjunction has requested payment, then there are two sources requesting payment, and the transaction is infeasible.

Returning to the example Section 6, we see the effect of payment requests. When B1 invokes the Send_Payment_or_Request (the last operation undertaken in the example from Section 6), it now executes the Request_Payment method on C. When the acquisition of D2, going on in parallel, also reaches this point, C will find in the Request_Payment call that another agent, B1, has already requested payment, therefore this transaction is doomed to failure.
12 Conclusion

In distributed network environments, what seems like a single sale to a customer may in fact be fulfilled by many sources and brokers contributing to a final information product. The customer wishes the whole sale will take place in an atomic fashion, so that if one part of the acquisition fails, none of the acquisitions succeed, and no money is spent. However, the sources have a different point of view. They are not concerned with the eventual disposition of documents that they sell, but expect to be compensated if they sell a document to a broker, even if that broker fails to generate a final sale to the customer. Therefore, to give the customer the desired semantics, but still provide protection to the brokers, the sequence of pairwise sales that lead to the full transfer must be carefully ordered.

We demonstrated a fully distributed algorithm that produces this ordering, and proved that it was sound (i.e., that any order that it generated had these “riskless” transaction properties) and that it was complete (so that if there were a riskless ordering, this algorithm would find it). We showed the operation of the algorithm on two examples, one where it found a riskless execution sequence, and one where a riskless sequence did not exist. We then considered extensions of the algorithm, first looking at direct trust between parties, so that under certain circumstances the safety conditions could be relaxed. Again, we showed how the algorithm could be modified to include this extension, and proved the extensions sound and complete. The next extension dealt with the issue of time, showing how a customer could impose a deadline by which the desired information must be received. The deadlines imposed by the customer percolated back through the system as each contributor to the final product imposed earlier deadlines on its sources so that all of the components would be in place in time for delivery to the customer. Once again, we showed that the modified algorithm was sound and complete for this formalization of time. Finally, we demonstrated three optimizations to the basic algorithm which enabled the customer to make an early detection of failures.
12.1 Implications

Multiple party transactions occur in many aspects of real life. Building contractors, wedding consultants, information brokers, and travel agents are all examples of professionals who facilitate transactions between multiple parties, and are paid for their services. Yet in the on-line world today, their electronic equivalents either do not exist or are treated externally to the transactions because mechanisms available for payment and structuring transactions are not readily available. Moreover, in the widely distributed on-line world, there is no guarantee that all of the parties to a transaction will be in the same jurisdiction or trust a single, universal intermediary. The framework described in this paper obviates the need for a universally trusted intermediary, instead enabling transactions where only pair-wise trusted intermediaries exist. This contribution leads the way for secondary markets to evolve in these brokerage services, providing greater opportunities for higher levels of service and value, and new profit-making opportunities for providers of these services. The formal properties of this framework provide a high degree of assurance that all of the parties in the transaction will profit from the interaction and will be protected from unscrupulous participants.

12.2 Future Work

This paper has laid considerable groundwork in the new area of distributed transactions. In the future, we hope to bring this level of formality to the device of indemnities, also introduced in [3]. Indemnities are a first step toward a full treatment of this material in a framework of expected utilities. In a decision theoretic framework, there may be cases when an agent will undertake a risky action (such as forwarding payment before being sure of receiving all the goods) when it has a positive expected value, because its potential rewards outweigh the costs of a poor outcome, taking into account the relative likelihood of the outcomes. The decision theoretic framework also allows more powerful distinctions about time. Rather than merely setting bounds, it is possible to express a distribution over the amount of time that each action might take. By chaining together the estimates for individual steps, agents may develop estimates for large pieces of the exchange. Combining these temporal estimates with the utilities of completing actions,
agents may rationally undertake unsafe actions for high rewards.

In this paper, we assumed that the cost of a document was always negligible compared to the agent’s available resources. If price considerations are added, certain transactions may be infeasible because a broker does not have sufficient funds to acquire the document before re-selling it. The trust mechanism described in this paper provides the tool to model solutions, with the customer advancing credit to the broker (sending money before the document is received) or the source offering a document on credit (before it receives payment from the broker). These basic concepts need to be integrated with the formal theory. A further assumption that was made in this discussion was that there was a single source for each document. When there are multiple sources, a transaction that fails in one step may not fail completely. Multiple sources also interact with utility values in interesting ways, since it may be worthwhile to order redundant copies of a document from different sources in order to ensure that all the pieces for a high value transaction will be in place when needed. There are also more complicated distributed transactions that do not always culminate in the sale to a single party. A more thorough study of the types of transactions and algorithms which allow safe execution of them in distributed systems is a significant area for new research.

References


13 Appendix

Code Fragment 1 is a description of the TaskRecord and SetRecord data structures used throughout the paper.

Code Fragment 1

TaskRecord:
Document: Document_ID
SourceAgent: Agent_ID /*ID of source party in the exchange*/
ClientAgent: Agent_ID
TrustedIntermediary: Agent_ID
Deadline: Time /*By which document must be received*/
Ult_Deadline: Time /*By which ultimate customer must receive goods*/
DocStatus: enumerated type {NOTHING, REQUESTED, SENT, RECEIVED, EXPIRED}
MoneyStatus: enumerated type {NOTHING, REQUESTED, SENT, RECEIVED, EXPIRED}
SRPointer: pointer to SetRecord

SetRecord:
Parent: pointer to SetRecord
ClientTR: TaskRecord
SourceTRs: Set of TaskRecord

End of Code Fragment 1

Code Fragment 2 presents the fully developed sound and complete algorithm for generating a safe execution sequence to obtain the documents in
a distributed transaction. The algorithm is distributed, with each principal
supporting all of the methods described in the top half of the figure. Trusted
intermediaries support those methods in the bottom half of the figure. The
behavior is completely event-driven, that is, agents are static, waiting for
method invocations to trigger action.

===

---

Code Fragment 2
---

Acquire_Set(SR):

/**<Called when the SetRecord has just been created, and fills in the TaskRecord
for each of the documents, either as already owned, or where to get it*/

    for all d in SR.SourceTRs
        if Has(d.Document):
            d.Source = self
            d.Trusted_Intermediary = None
            d.DocStatus = RECEIVED
        else:
            if no known source:
                (SR.Client.ClientAgent).unknown_document(SR.client)
            else:
                d.Source = relevant_source
                d.TrustedIntermediary = relevant_trusted_intermediary
                d.Deadline = SR.client.deadline -
                DELIVERYTIMETO(SR.client.ClientAgent)
                d.Ult_Deadline = SR.client.Ult_Deadline
                d.DocStatus = NOTHING
                d.MoneyStatus = NOTHING
                self.check_for_next(SR)

Check_for_next(SR)

/**<The dispatch routine that decides on the next step to take. It considers
five different cases:*/

    1) The ultimate customer’s deadline has expired, so there’s no way the
       transaction can be completed. Call it off & return everything obtained on
       trust.

    2) The agent has all the desired documents, so it’s either done or can
       send them on to its customer. If some of the documents were acquired on
       trust, they either have to be paid for or returned.

    3) Trusted intermediaries have all the pieces the agent doesn’t—the agent
       can send money to acquire them if his customer has set aside the money.
       Otherwise, the agent should request money from the customer, so he is not
       risking his own money.
4) If one document hasn’t been acquired yet, and the customer has set aside money, then the agent can use its own resources to obtain the last missing conjunct. Otherwise, request money from the customer.

5) If none of the previous conditions hold, the agent should ask the sources to send the documents (without promising anything in return.)

*/
/*CLAUSE 1 - Ultimate Deadline passed */
if NOW > SR.client.Ult_Deadline:
    self.expire(SR)
else:
/*CLAUSE 2 - all pieces in hand*/
if all SR.SourceTRs.DocStatus is RECEIVED:
    if SR.client.ClientAgent == self: //if you’re the ultimate customer*/
        for all d in SR.SourceTRs: /*Buy all docs received on trust*/
            if d.MoneyStatus != SENT or RECEIVED:
                P = Payment Information
                d.SourceAgent.Receive_Payment(P,d,SR)
                d.MoneyStatus = SENT
                self.success(SR)
            elseif SR.client.DocStatus != SENT /*send on to customer*/
                self.send_Doc(SR)
/*CLAUSE 3 - all pieces are held or at trusted intermediaries*/
if all SR.SourceTRs.DocStatus == SENT or RECEIVED:
    self.send_All_Payments_or_Request(SR)
/*CLAUSE 4 - One missing*/
if some SR.SourceTRs.DocStatus == NOTHING or REQUESTED and all other SR.SourceTRs.DocStatus == SENT or RECEIVED
    self.send_Payment_or_Request(SR)
/*CLAUSE 5 - more than one missing*/
elseif some SR.SourceTRs.DocStatus == NOTHING or REQUESTED:
    for all d in SR.SourceTRs:
        if d.DocStatus == NOTHING:
            d.DocStatus = REQUESTED
            (d.SourceAgent).Request_Doc(d, SR)
send_Doc(SR):
/* Called to determine the proper way to send the document to the customer. Send directly or via a trusted intermediary*/
SR.client.DocStatus = SENT
if trusted(SR.client.ClientAgent) or SR.client.MoneyStatus = RECEIVED or SR.client.Cost == 0:
else:
    (SR.client.TrustedIntermediary).Receive_Doc(compose(SR.SourceTRs),
    SR.client, SR, self)

expire(SR):
/* Called when an exchange has already reached its expiration point*/
for all d in SR.SourceTRs: /* Return all docs received on trust*/
    if d.DocStatus != RECEIVED:
        d.SourceAgent.Return_Document(text(d), d)
    if d.DocStatus != EXPIRED
        d.DocStatus = EXPIRED
        d.SourceAgent.expire(d.SRPointer)
        d.TrustedIntermediary.expire(d)
    if d.client.MoneyStatus == RECEIVED & (d.client.DocStatus == SENT or
        d.client.DocStatus == RECEIVED):
        SR.client.ClientAgent.Return_Money(P, d, SR)
    if SR.client.DocStatus != EXPIRED:
        SR.client.ClientAgent.expire(d.SRPointer.Parent)
        SR.client.TrustedIntermediary.expire(d.SRPointer.Parent.client)
    if SR.client.ClientAgent == SR.SourceTRs.ClientAgent: /*ultimate customer*/
        self.fail(SR)

Receive_Doc(D, TR, SR, sender):
/* Allows the recipient of a document to activate dispatch routine to figure
out what to do next*/

    store D
    if TR.SourceAgent == sender & TR.MoneyStatus == SENT:
        /* Source trusts customer, sent doc directly, but customer sent money
to TI. Let TI know to give money to source. */
        TR.TrustedIntermediary.release_Payment(TR, SR)
        self.check_for_next(TR.SRPointer.Parent)

Notify_Doc(TR):
/* Allows the requester of a document to activate dispatch routine when the
desired document has arrived at the trusted intermediary*/

    TR.DocStatus = SENT
    self.check_for_next(TR.SRPointer.Parent)

Receive_Payment(P, TR, SR):
/* Allows recipient of cash to activate dispatch routine to figure out what to
do next. P is the payment specific information, TR is the TaskRecord status
to update, SR is unused, but included to maintain consistency with the trusted
intermediary which needs the SR when invoking notify_payment on the source.
*/
Add P to cash
TR.MoneyStatus = RECEIVED
for all d in SR.SourceTRs where d.DocStatus == RECEIVED:
//*Pay for all docs you got on trust*/
if d.MoneyStatus != SENT or RECEIVED:
P = Payment Information
d.SourceAgent.Receive_Payment(P, d, SR)
d.MoneyStatus = SENT
check_for_next(TR.SRPointer)

Send_All_Payments_or_Request(SR):
//Called when all unowned conjuncts are at the TI’s. (if money from customer
has been set aside, or request it if not).* /
if check_time(SR):
if SR.client.MoneyStatus == SENT:
for all d in SR.SourceTRs
if d.MoneyStatus != SENT:
d.MoneyStatus = SENT
P = Payment Information
if trusted(d.sourceAgent):
   (d.sourceAgent).Receive_Payment(P, d, SR)
else:
   (d.TrustedIntermediary).Receive_Payment(P, d, SR)
else:
   SR.client.MoneyStatus = REQUESTED
(SR.client.ClientAgent).Request_Payment(SR.client)
else:
   self.expire(SR)
   if SR.client.ClientAgent == SR.client.SourceAgent: /*ult. customer*/
   self.not_in_time(SR)

Send_Payment_or_Request(SR):
//This routine should be called if exactly one document is not at TI. It
either sends money to one TI for missing conjunct or requests the document
and payment (or there’s an expiration failure)*/
for all d in SR.SourceTRs:
if d.DocStatus != SENT and d.DocStatus != RECEIVED
   missing_doc = d /*This conjunct is still needed*/
missing_doc.DocStatus = REQUESTED
(missing_doc.SourceAgent).Request_Doc(missing_doc, SR)

if SR.client.MoneyStatus == SENT or RECEIVED and check_time(SR):
/* do an unconditional obtain*/
missing_doc.MoneyStatus == SENT
P = Payment Information
if trusted(missing_doc.sourceAgent):
   (missing_doc.sourceAgent).Receive_Payment(P, missing_doc, SR)
else:
(missing_doc.TrustedIntermediary).Receive_Payment(P, missing_doc, SR)

else:
    if not check_time(SR):
        self.expire(SR)
        if SR.client.ClientAgent == SR.client.SourceAgent: /*ult. cust*/
            self.not_in_time(SR)

        if SR.client.MoneyStatus == NOTHING: /*request money*/
            SR.client.MoneyStatus = REQUESTED
            (SR.client.ClientAgent).Request_Payment(SR.client)

check_time(SR):
    /*Called to determine if document will arrive in time, guaranteed. In, even
    if all transit times take their worst case upper bound (DELIVERYTIMETO) the
    document will still arrive before the customer’s deadline. If the transaction
    can’t be completed in time, report failure, otherwise allow processing to
    continue. Checks that the latest arriving conjunct will still allow you to
    send it to the customer before his deadline. Assumes: 1) No processing time;
    2) All delivery times are symmetric; 3) Times for trusted intermediaries are
    included in the DELIVERYTIMETO*/

    MaxDeliveryTime = 0
    for all d in SR.SourceTRs:
        if d.DocStatus == SENT:
            MaxDeliveryTime = max(DELIVERYTIMETO(d.trustedIntermediary),
                                   MaxDeliveryTime)
        elseif d.DocStatus != RECEIVED:
            MaxDeliveryTime = max(DELIVERYTIMETO(d.sourceAgent),
                                   MaxDeliveryTime)

    if NOW + MaxDeliveryTime +
       DELIVERYTIMETO(SR.client.ClientAgent) > SR.client.Deadline:
        return false
    else
        return true

Notify_Payment(TR, hisSR):
    /*Called when a trusted intermediary receives payment from a customer, to warn
    the source that it is guaranteed to get payment when the desired documents are
    provided.*/

    if TR.SRPointer != NULL:
        /*record was created previously by a Request_Document call*/
        TR.client.MoneyStatus = SENT
        self.check_for_next(TR.SRPointer)
    else: /*This is a new request, create a new SR*/
        SR = new SetRecord
        SR.client = TR
        SR.client.SRPointer = SR /*If this is not pass by reference, get
client to update his, too*/

SR.Parent = hisSR
SR.SourceTRs = decompose(TR.document)
self.acquire_set(SR)

Request_Document(TR, hisSR):
/*Called when an agent is asked to provide a document (with no guarantee of payment). Creates a SetRecord and calls the dispatcher.*/

SR = new SetRecord
SR.client = TR
SR.client.SRPointer = SR  /*If this is not pass by reference, get client to update his, too*/
SR.client.DocStatus = REQUESTED
SR.SourceTRs = decompose(TR.document)
SR.Parent = hisSR
self.acquire_set(SR)

Request_Payment(myTR):
/*Called when a source needs payment in order to proceed with obtaining the desired documents. [Note that even if payment is provided, there is no guarantee that the document will be obtained--a refund might be issued instead.] This request is basically a no-op. The customer will already proactively send payment whenever it can, so a request does nothing except give a status update to the customer. The customer knows that if there are two requests for payment, the transaction can not be completed risklessly.*/

for d in myTR.SRPointer.SourceTRs:
  if d.MoneyStatus = REQUESTED:  /*if any other conjunct requested*/
    /*This acquire will fail, due to buying 1/2 conjunction risk*/
    self.expire(myTR.SRPointer)
myTR.MoneyStatus = REQUESTED

unknown_document(TR):
/*Called when the source can't find a way to provide the document*/

self.expire(myTR.SRPointer)

Return_Payment(P, TR):
/*Receive cash back, will be followed by call to expire*/

Return_Document(D, TR):
/*Receive document back, will always be followed by call to expire*/

TRUSTED INTERMEDIARY

/*The trusted intermediaries are store some information that would be available from TaskRecords that are passed as parameters with the method calls. So as not to rely on this information which is provided by interested
participants, the trusted intermediaries use their own simple record keeping mechanism. When a trusted intermediary receives a document, it stores the contents of the document as text(Doc_ID), and sets Has(TR) to true for that TR. If the payment arrives before the document, then paid_for(TR) is set to true, and the payment information is stored as payment(TR).

Receive_Payment(P, clientTR, clientSR)
/*Called by a customer to send payment to a source, via a trusted intermediary. The trusted intermediary checks to see if it already has the document in question and if so swaps them, otherwise, it stores the payment information and notifies the source that the money is there whenever the document arrives.*/

if NOW + DELIVERYTIMETO(clientTR.clientAgent) > clientTR.deadline:
   self.Check_for_expirations()
else Has(clientTR):
   /*checks not only doc, but source & dest*/
   (clientTR.sourceAgent).Receive_Payment(P, clientTR, clientSR)
   (clientTR.clientAgent).Receive_Doc(text(clientTR.document), clientTR, clientSR, self)
else:
   paid_for(clientTR) = true
   payment(clientTR) = P
   (clientTR.sourceAgent).Notify_Payment(clientTR, clientSR)

Receive_Doc(D, clientTR, clientSR, sender)
/*Called by a source to send a document to a customer, via a trusted intermediary. The trusted intermediary checks to see if it already has the payment in question and if so swaps them, otherwise, it stores the document and notifies the customer that the document is there whenever the money arrives.*/

if NOW + DELIVERYTIMETO(clientTR.clientAgent) > clientTR.deadline:
   self.Check_for_expirations()
else paid_for(clientTR):
   clientTR.client.Receive_Doc(D, clientTR, clientSR, self)
   clientTR.source.Receive_Payment(payme(clientTR), clientTR, clientSR)
else:
   Has(clientTR) = true
   text(clientTR) = D
   clientTR.client.Notify_Doc(clientTR, clientSR)

release_Payment(clientTR, clientSR)
/*Called by a customer who had previously sent money to TI, then received document directly from source. Allows the TI to send payment on immediately to the source.*/

clientTR.source.Receive_Payment(payme(clientTR), clientTR, clientSR)

Check_for_expirations()
/*Called periodically (or before any exchange known to expire) to abort...*/
exchanges which have reached the expiration date. Notify client and return the appropriate pieces.*/

for all d in TR's:
    if NOW + DELIVERYTIMETO(d.ClientAgent) > d.Deadline:
        (d.ClientAgent).not_in_time(d)
        if paid_for(d):
            (d.clientAgent).Return_Payment(payment(d), d)
        (d.clientAgent).expire(d.SRPointer.Parent)
    if Has(d):
        (d.SourceAgent).Return_Document(text(d), d)
        (d.SourceAgent).expire(d.SRPointer)

expire(TR)
//Called by one of the principals if this exchange cannot be completed. The principal will handle notifying the other principals. The TI should just return the goods to their owners.*/

if paid_for(d):
    (d.clientAgent).Return_Payment(payment(d), d)
    (d.clientAgent).expire(d.SRPointer.Parent)
if Has(d):
    (d.SourceAgent).Return_Document(text(d), d)
    (d.SourceAgent).expire(d.SRPointer)

End of Code Fragment 2
==============================================================================