Distributed Constraint Reasoning under Unreliable Communication

Pragnesh Jay Modi, Syed Muhammad Ali, Rishi Goel, Milind Tambe
University of Southern California/Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292, USA
{modi@isi.edu, syedmuha@usc.edu, rishi.goel@attws.com, tambe@usc.edu}

Abstract

We investigate how algorithms for Distributed Constraint Reasoning (DCR) can be modified to operate effectively over unreliable communication infrastructure. While DCR algorithms typically assume that communication is perfect, this assumption is problematic because unreliable communication is a common feature of many real-world multiagent domains. Limited bandwidth, interference, loss of line-of-sight are some reasons why communication can fail. We introduce a novel method for dealing with message loss in the context of a particular DCR algorithm named Adopt. The key idea in our approach is to let the DCR algorithm inform the lower error-correction software layer which key messages are important and which can be lost without significant problems. This allows the algorithm to flexibly and robustly deal with message loss. Results show that with a few modifications, Adopt can be guaranteed to terminate with the optimal solution even in the presence of message loss and that time to solution degrades gracefully as message loss probability increases. The results also suggest that artificially introducing message loss even when communication infrastructure is reliable could be beneficial in terms of the amount of work agents need to do to find the optimal solution.

Introduction

Distributed Constraint Reasoning (DCR) allows a set of multiple agents to solve problems in a decentralized manner through the communication of peer-to-peer messages. It has been proposed as a way to model and solve many distributed problems that arise in multiagent systems, with a variety of algorithms having been recently developed (Modi *et al.* 2003) (Zhang & Wittenburg 2002) (Yokoo 2001). In order to provide strong guarantees on the correctness and completeness of their algorithms, algorithm designers have typically made the following two assumptions about the communication infrastructure on which the distributed algorithm operates:

- *Reliable*: Delay in delivering a message is finite, i.e., every message sent is eventually received.
- *Atomic*: Order is preserved in transmissions between any pair of agents.

In this paper, we consider how we can relax the reliability assumption without giving up key algorithm properties such as the guarantee of termination with a correct solution. We assume a simple form of unreliable communication: the communication infrastructure has an unknown *loss probability* $r < 1$, where a message is dropped (not delivered) with probability r. In our experiments we will assume r is constant over time, but this is not strictly necessary. Investigation for relaxation of the atomic assumption is an issue for future work.

A common method for dealing with unreliable channels in communication networks is to implement an error correction layer in software that can ensure reliable message delivery even when the communication infrastructure itself is inherently unreliable. This is typically done through an acknowledgment protocol where ACK messages are used to verify that a message has been received. A number of such protocols have been developed in the field of computer networking, the most popular of which is TCP (Stevens 1994). Unfortunately, recent research provides evidence that TCP is infeasible in many types of communication networks important for applications in multiagent systems, such as wireless and ad-hoc networks (Balakrishnan *et al.* 1996) (Padhye *et al.* 1998).

Simply relying on a lower layer error-correction mechanism to ensure reliable delivery is an inadequate approach for dealing with unreliable communication infrastructure when developing multiagent algorithms. First, it can significantly increase the number of messages that must be communicated since every message must be acknowledged. Second, a sender cannot send any messages to a given agent x_i until the ACK for a previously sent message is received from x_i for fear of the violating the atomic assumption mentioned earlier. The time cost in waiting for ACKs can degrade performance and reduce the efficiency of the higher-level DCR algorithm. Third, this method is unable to take advantage of the fact that it may be okay for some messages to be lost without large negative effects on the higher-level DCR algorithm.

In this paper, we propose a novel approach to dealing with message loss in DCR algorithms. We assume the availability of an asynchronous DCR algorithm and a lower error-correction software layer. Instead of relying exclusively on the error-correction layer, we advocate giving the DCR al-

gorithm itself the control to decide which messages must be communicated reliably and which can be lost. The idea is that by requiring only key messages be communicated reliably while allowing other messages to be lost, we can design DCR algorithms that are more flexible and robust to message loss. We use the Adopt algorithm for the Distributed Constraint Optimization Problem (DCOP) as a vehicle for our investigation. Adopt is a decentralized optimization algorithm that can provide strong guarantees on the global quality of its solutions. This algorithm is suitable for our investigation for two reasons. First, the algorithm is completely asynchronous which allows agents to continue processing messages even when some messages are lost. Second, the algorithm has a built-in termination detection mechanism. This allows agents to automatically detect deadlock situations that may arise due to message loss.

We show how our ideas can be implemented within the context of the Adopt algorithm and the benefits that are derived. With a few modifications, we show that Adopt is still guaranteed to terminate with the optimal solution even if communication is unreliable. Experimental results show that Adopt's performance degrades gracefully with message loss. We also present results that suggest that artificially introducing message loss even when communication is reliable could be a way to decrease the amount of work agents need to do to find the optimal solution. Indeed, recent work by Fernandez et al. has shown that artificially introducing communication delay in DCR can have beneficial effects on the performance of DCR algorithms.

DCOP Definition

A Distributed Constraint Optimization Problem (DCOP) consists of n variables $V = \{x_1, x_2, \ldots x_n\}$, each assigned to an agent, where the values of the variables are taken from finite, discrete domains $D_1, D_2, \ldots, D_n$, respectively. Only the agent who is assigned a variable has control of its value and knowledge of its domain. The goal is for the agents to coordinate their choice of values so that a global objective function is optimized. This function is modeled as a set of valued constraints between variables. Each agent knows only about the constraints in which its variable is involved. In this paper, we will assume each agent has a single variable and use the terms "agent" and "variable" interchangeably.

More formally, for a pair of agents x_i, x_j, there may exist a valued constraint $f_{ij} : D_i \times D_j \rightarrow N \cup \infty$ between their variables. We assume only binary constraints. If a constraint exists between a pair of agents, we call them *neighbors*. Figure 1.a shows an example constraint graph with four agents. Although all constraints are identical in the example, this is not required. The objective is to find an assignment of values to variables such that the total cost, denoted F, is minimized and every variable has a value. For example in Figure 1.a if all variables are assigned the value 0 then $F = 4$. If all variables are assigned the value 1 then $F = 0$ which is the optimal solution.

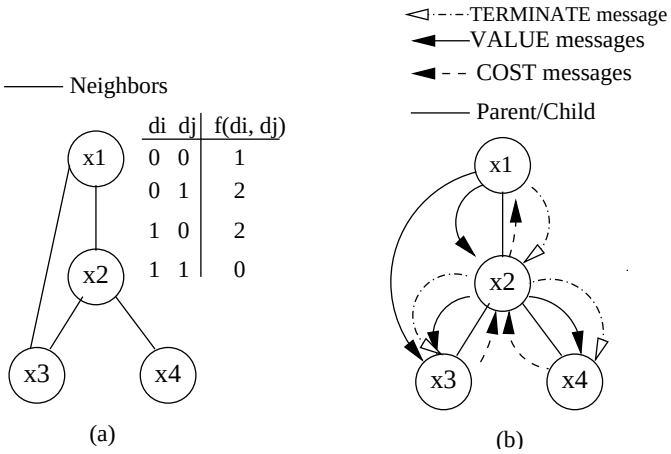

Figure 1: (a) Constraint graph. (b) Communication graph.

Adopt Algorithm for DCOP

We briefly describe the Adopt algorithm for DCOP and refer the reader to (Modi *et al.* 2003) for details.

In Adopt, agents are prioritized in a Depth-First Search (DFS) tree in which each agent has a single *parent* and multiple *children*. Figure 1.b shows a DFS tree formed from the constraint graph in Figure 1.a – x_1 is the root, x_1 is the parent of x_2 and x_2 is the parent of both x_3 and x_4. Constraints are allowed between an agent and any of its ancestors or descendents (there is a constraint between x_1 and x_3), but there can be no constraints between nodes in different subtrees of the DFS tree. We assume parent and children are neighbors. These requirements place no restrictions on the constraint network itself – every connected constraint network can be ordered into some DFS tree(Lynch 1996).

Once the DFS tree is constructed, each agent x_i concurrently executes the following algorithm.

- Initialize the lower bound for each value in D_i to zero. Assign a random value to x_i.
- Send current value of x_i to each neighbor lower in the DFS tree.
- When receive the value of a neighbor x_j, for each value D_i evaluate the constraint between x_i and x_j. Add the cost of the constraint to the lower bound for each value. If the lower bound for the current value is higher than the lower bound for some other value d, switch value to d.
- Send the lower bound for the value with least lower bound to parent in the DFS tree. Attach the variable value of the parent under which this lower bound was computed.
- When receive a lower bound from child attached with value d, add the reported lower bound to the lower bound for d. If the lower bound for the current value is higher than the lower bound for some other value d, switch value to d.
- When a higher neighbor changes variable value, reinitialize the lower bound for each value in D_i to zero.
- Continue sending and receiving messages and changing values as dictated above until the following *termination*

condition is true: The lower bound LB for one value d is also an upper bound, and the lower bound for all other values is higher than LB. Note that when this condition is true d is the globally optimal value for x_i until and unless a higher neighbor changes value.

The communication in Adopt is shown in Figure 1.b. Variable values are sent down constraint edges via VALUE messages. A COST message is sent only from child to parent. (Another type of message not discussed here is a THRESHOLD message which is used to increase efficiency.)

Once the above termination condition is true at the root agent, the root sends a TERMINATE message to its children and terminates itself. After receiving a TERMINATE message, an agent knows that all of its higher neighbors have terminated. Once the termination condition is true at non-root agent x_i and it has received a TERMINATE message from its parent, x_i will send TERMINATE messages down to its children. In this way, TERMINATE messages are recursively sent down the tree until the termination condition is true at all agents and all agents have terminated. The Adopt algorithm has been proven to terminate with the globally optimal solution (Modi *et al.* 2003), i.e., it is sound and complete assuming reliable communication.

Algorithm Modifications for Message Loss

We now consider the effect of message loss on the Adopt algorithm described above. Since Adopt is completely asynchronous, it is well suited for operating under unreliable communication infrastructure. In particular, agents are able to process any message no matter when it is received and are insensitive to the order in which messages are received. This is in contrast to synchronous algorithms which require messages to be received and sent in a particular order. For example in synchronous algorithms such as Synchronous Branch and Bound (Hirayama & Yokoo 1997), if a message is lost no progress can be made until that message is successfully retransmitted.

While asynchrony is a key advantage, the major difficulty that arises is the danger of deadlock. Deadlock can occur for two reasons: the loss of VALUE/COST messages, or the loss of TERMINATE message. We consider each case separately. If VALUE or COST messages are lost, we could have a deadlock situation with agents waiting for each other to communicate. For example, consider two agents x_1 and x_2. x_1 sends a VALUE message to x_2. x_2 evaluates and sends back to x_1 a COST message. Suppose this message gets lost. At this point, x_1 is now waiting for a COST message from x_1, while x_2 is waiting for another VALUE message from x_1. Thus, the loss of one message has resulted in the agents getting deadlocked.

It turns out that we can overcome deadlock problems arising from loss of VALUE and COST messages due to another key novelty of the Adopt algorithm: Adopt's built-in termination detection. The termination condition allows an agent to locally determine whether the algorithm has terminated. If no messages are received for a certain amount of time and an agent's termination condition is not true, then that agent can conclude that a deadlock may have occurred due to message loss. The agent can then resend VALUE and COST messages to its neighbors in order to trigger the other agents and break the deadlock. This method requires the implementation of a timeout mechanism at each agent. The value of the timeout can be set according to the average time between successive messages. This solution is more flexible and efficient than the alternative approach of dealing with message loss at a lower error-correction software layer. Instead, the algorithm intelligently determines when messages need to be resent by using the local termination condition as a guide. This method is also stateless in the sense that an agent does not need to remember the last message it sent in case a resend is needed. The agent can simply send out its current value and current cost whenever a timeout occurs.

Another reason deadlock can occur is if a TERMINATE message is lost. Agents terminate execution in response to the TERMINATE message received from their parents. If that message gets lost, an agent has no knowledge that the rest of the agents have come to a stop. The TERMINATE messages are essentially a distributed snapshot mechanism (Chandy & Lamport 1985) whereby the agents determine that the termination condition is true at all the other agents. Unfortunately, distributed snapshot algorithms require reliable communication. Thus, Adopt requires that TERMINATE messages be sent reliably. This is can be done through an acknowledgment protocol where each TERMINATE message sent must be acknowledged by the recipient by responding to the sender with an ACK message. The sender will resend its message if an ACK is not received after certain amount of time. The sender continues to resend until an ACK is eventually received. Since $r < 1$, the TERMINATE message and the ACK will eventually (in the limit) go through. Since TERMINATE messages only need to be communicated once between parent and child, the overhead for this is not very severe, and is certainly less than requiring every message to be communicated reliably.

With these modifications it can be ensured that Adopt will eventually terminate with the optimal solution, regardless of the amount of network disturbance, so long as the probability of a message being lost is less than 1. To see that Adopt will eventually terminate, realize the deadlock detection timeout will ensure that an agent x_i will not sit waiting forever for a message that may not come when its own termination condition is not true. Instead, x_i will continue sending messages to its children until a reply is received. Thus, each child will eventually report to x_i a lower bound that is also an upper bound. When this occurs, x_i's termination condition will finally be true and it can terminate.

Experiments

We experiment on distributed graph 3-coloring. One node is assigned to one agent who is responsible for its color. Global cost of solution is measured by the total number of violated constraints. We experiment with graphs of link density 3 – a graph with link density d has dn links, where n is the number of nodes in the graph. The randomly generated instances were not explicitly made to be overconstrained, but note that

Table 1: Algorithm comparision for zero loss

	Time(sec)		Messages	
	Orig Adopt	Modified Adopt	Orig Adopt	Modified Adopt
8 Agents	171.17	160.12	35038	35002
10 Agents	213.19	203.4	41809	42085
12 Agents	636.29	622.12	114211	115939

link density 3 is beyond phase transition so randomly generated graphs with this link density are almost always over-constrained. We average over 6 randomly generated problems for each problem size and each problem was run three times for each loss probability, for a total of 18 runs for each datapoint. Each agent runs in a separate thread and time to solution is measured as CPU time. We use a uniform time-out value of 10 seconds. We ensured consistent system load between runs and each run produced an optimal solution.

Table 2 shows the relative change in running time as a percentage of the running time when communication is perfect ($r = 0$). We see that as loss probability r increases from 0% to 10%, the running time increases very little – only 5.88% for 10 agents and 4.66% for 12 agents. At loss probablity of 20%, we begin to see more severe effects on running time – 20.95% for 10 agents and 19.31% for 12 agents. The data provides initial evidence that Adopt's performance degrades gracefully as message loss rate increases. It is interesting to note that solution time decreases slightly at 2% loss probability as compared to 0%. This suggests that perhaps message loss can be beneficial, but more analysis is necessary.

In addition to solution time, we would also like to know if the agents are doing more or less work when messages are being lost as compared to when communication is perfect. One measure of "work" is the total number of messages processed. Table 3 shows the relative change in the total number of messages processed as a percentage of the number of messages processed when communication is perfect. We see that agents process fewer messages as message loss rate increases – around 8% less for 12 agents at 20% loss. These results show that agents are able to obtain a solution of optimal quality but by processing fewer messages as compared to the perfect communication case. This suggests that artificially introducing message loss even when communication is reliable could be a way to decrease the amount of work agents need to do to find the optimal solution. In fact, recent work by Fernandez et al. has shown that artificially introducing communication delay in DCR can have beneficial effects on the performance of DCR algorithms (Fernandez *et al.* 2002). This is something we will explore in future work.

Table 2: Pct change in running time with increasing loss rate

Loss rate(r)	8 Agents	10 Agents	12 Agents
0%	100.00%	100.00%	100.00%
2%	99.61%	98.84%	100.17%
5%	103.94%	100.40%	100.01%
10%	110.78%	105.88%	104.66%
20%	128.93%	120.95%	119.31%

Table 3: Pct change in number of messages processed (rcvd) with increasing loss rate

Loss rate(r)	8 Agents	10 Agents	12 Agents
0%	100.00%	100.00%	100.00%
2%	98.80%	98.01%	98.36%
5%	98.44%	96.47%	95.27%
10%	98.63%	95.04%	93.02%
20%	97.58%	92.24%	92.59%

Conclusions

We found that while sending acknowledgements for every message was excessive and too expensive, and sending none was problematic, the right tradeoff was to allow the DCR algorithm to control which key messages need to be sent reliably. We showed that this method allows an asynchronous algorithm to tolerate message loss and still terminate with the globally optimal solution. Empirical results showed that time-to-solution increased gradually as message loss rate is increased which is a desireable property in a DCR algorithm. We also found that agents need to process fewer messages to find the optimal solution when messages may be lost, which suggests that an active loss mechanism may improve algorithm performance.

References

Balakrishnan, H.; Padmanabhan, V.; Seshan, S.; and Katz, R. 1996. A comparison of mechanisms for improving tcp performance over wireless links. In *Proceedings of ACM SIGCOMM*.

Chandy, K., and Lamport, L. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*.

Fernandez, C.; Bejar, R.; Krishnamachari, B.; and Gomes, C. 2002. Communication and computation in distributed csp algorithms. In *Principles and Practice of Constraint Programming*.

Hirayama, K., and Yokoo, M. 1997. Distributed partial constraint satisfaction problem. In Smolka, G., ed., *Principles and Practice of Constraint Programming*. 222–236.

Lynch, N. 1996. *Distributed Algorithms*. Morgan Kaufmann.

Modi, P. J.; Shen, W.; Tambe, M.; and Yokoo, M. 2003. An asynchronous complete method for distributed constraint optimization. In *Proc of Autonomous Agents and Multi-Agent Systems*.

Padhye, J.; Firoiu, V.; Towsley, D.; and Krusoe, J. 1998. Modeling tcp throughput: A simple model and its empirical validation. In *ACM Computer Communications Review: Proceedings of SIGCOMM 1998*.

Stevens, W. R. 1994. *TCP/IP Illustrated, Volume 1*. Addison-Wesley.

Yokoo, M. 2001. *Distributed Constraint Satisfaction:Foundation of Cooperation in Multi-agent Systems*. Springer.

Zhang, W., and Wittenburg, L. 2002. Distributed breakout revisited. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*.