Chapter 20

Distributed Systems Issues

When you have finished giving the king
these details of the battle,..
Joab

This thesis has presented a set of asynchronous protocols and algorithms for solving distributed CSPs. We have proven their completeness, correctness and termination. Several contributions of this thesis are applications of a marking technique, signatures, that we have initially developed for the Asynchronous Aggregation Search (AAS). As proven in Chapter 8, signatures offer causal order within families of messages with patterns of exchanges in asynchronous backtracking.

When a distributed application has to be built based on protocols presented in this thesis, one needs to implement several other techniques for termination or solution detection. Many authors in the field have used only simulators and were not particularly interested in developing custom techniques. Nevertheless, other authors have worked with real implementations and have employed general techniques (Hamadi 1999b). In our work, we have preferred to research possible alternatives to the general techniques.

20.1 Termination Detection

In the several distributed algorithms we need to detect the quiescence of the agents. In previous work, distributed CSP algorithms have detected quiescence using the techniques for distributed snapshot presented in (Chandy & Lamport 1985). That algorithms have the characteristic that termination tests need to be initialized by agents suspecting quiescence. If the interest is to detect quiescence as quickly as possible, other methods are more appropriate.

We present here a termination detection method which only requests one sequential message after the quiescence of the monitored protocol. This technique is well adapted to consistency maintenance procedures for distributed CSPs. Related termination detection protocols are already known to the distributed systems community (Kumar 1985; Mattern 1987) and their proof also applies here.

Each agent A_i maintains a counter $c_{i,j}$ for outgoing messages towards each other agent A_j and a counter $c^{j,i}$ for incoming messages from each agent A_j. When A_i becomes idle, this counters are sent at any modification to the agent T checking the termination. When T detects that $c_{i,j} = c^{i,j}$ for all i and j, then T can announce termination. Since the counters can only increase, there is no need of time stamps or FIFO channels since the highest counter value is always the newest. Several termination detection stages can succeed synchronously one after another, as happens with the successive consistency rounds in sMDC. We may want to distinguish them. A simple flag with two values that switches at the start of a new round has to be attached to each message. If each agent receives a message in each round, this is sufficient to announce the agents that a new stage begins and that the local counters must be reset to 0. Otherwise the flag has to be replaced by an increasing counter of the stages.

The termination detection messages need not be sent directly to T. Each agent keeps a list Lc of counters that it has received. Lc contains only the last received pair of vectors of counters for each agent. If A_i has to send messages to a set of agents A, it chooses an agent A_j in A. A_i attaches to the message sent to A_j the modified counters in Lc, received from other agents, as well as and its own updated counters. Then A_i clears Lc. If A was empty after A_i has received some messages, A_i sends its counters and its Lc to T and also clears Lc. When Lc is large enough to fill the payload of a message, the network charge is reduced if an agent can sent the modified counters to T rather than sending them further to other agents. The size of Lc when the behavior has to change is a function of the size $|m|$ of the messages sent to agents in A and also depends on the maximum unfragmented payload (MTU) that can be sent to T. The threshold should be $|Lc| = (MTU - |m| mod\ MTU) mod\ MTU$. The second *modulo* operation is needed for translating MTU into 0.

20.2 Solution Detection

In the described implementations of search protocols for DisCSPs, solutions are only detected upon quiescence[1]. This state is usually recognized using general purpose distributed mechanism (Chandy & Lamport 1985). We have noticed that in the particular case of asynchronous search, solutions can be detected before quiescence. This means that termination can be inferred earlier and that the number of messages required for termination detection can be reduced.

20.2.1 Static ordering on agents

We start describing the solution detection technique used in the Asynchronous Aggregation Search. When an agent has proposed a solution to its local problem that satisfies its view, it is said to be in the state `Solution` as long as it receives no refusal of that proposal.

Definition 20.1 *We call* end agent *an agent that has no incoming link.*

The *composition of two partial valuations*, v_1, v_2, consists in a valuation, v, $v{=}v_1{\cap}v_2$. If x_i is assigned in both v_1 and v_2 then v is non-empty only if x_i is assigned with the same value in v_1 and in v_2. We have introduced a system message (not considered in the notion of quiescence) called **accepted** which informs the sender of an **ok?** message of the acceptance of its proposal:

- **accepted** messages are sent from an agent to all its predecessors (along all incoming links). If the agent has been an end agent, it also sends an **accepted** to the *system agent*,
- an **accepted** message has as parameter a Cartesian product obtained by intersecting the current instantiation of the sender with the parameters of the last **accepted** messages received from all its outgoing links[2],
- an **accepted** message is sent by an agent only when its parameter is non empty (i.e does not contain empty domains), all the outgoing links have presented an **accepted** message and the agent is in the state `Solution`,
- the agents checks whether to send **accepted** messages when they reach the state `Solution` or when they receive **accepted** messages.
- **accepted** messages are FIFO ordered, (e.g. using additional counters).

Let D_i be the subgraph induced by the agents A^j with $j > i$ such that A^j can be reached from A^i along the directed links initialized by the *system agent*.

[1] End of **ok?**, **nogood** and **add-link** messages.

[2] We define the intersection $S_i \cap S_j$ of two Cartesian products S_i and S_j as the Cartesian product of the union of all variables implied in S_i and S_j. The domain of each variable of $S_i \cap S_j$ is given by the intersection of its domains in S_i and S_j.

Proposition 20.1 *If a given agent A^i receives an* **accepted**(S_k) *message from all its outgoing links k, and if $\bigcap_k S_k \neq \emptyset$, then A^i can infer that $\bigcap S_k$ is a solution for the partial CSP defined by the agents of D_i.*

Proof sketch. A very detailed proof is available at (Silaghi 2000b). D_i is a directed acyclic graph. If a given node A^j of this graph receives an **accepted**(S_k) message from all its k direct successors such that $\bigcap S_k \neq \emptyset$, it is obvious that the k successors have found an agreement on all the elements of $\bigcap S_k$. Following the definition of **accepted** messages, the agent A^j can in turn send an **accepted** through all its incoming links and the process be repeated recursively. The proposition is therefore simply proved by induction on D_i. □

Corollary 20.1.1 *A correct solution is detected when the system agent receives an* **accepted**(S_i) *message from each initial end agent A^i and when $\bigcap_i S_i \neq \emptyset$.*

The method used for termination detection in AAS has been later (Silaghi *et al.* 2000g) improved by sending **accepted** messages only on arcs forming a spanning tree of the directed graph on which they were sent in (Silaghi *et al.* 2000a). We choose only one arc along which an agent sends **accepted** messages. This arc is chosen among initial links and leads to the nearest priority agent among those to which **accepted** messages were sent in (Silaghi *et al.* 2000a).

For ABT, these techniques can be adapted, treating the corresponding protocol as an equivalent of AAS where agents only own the constraints that they enforce.

20.2.2 Dealing with agent reordering

The solution detection technique in (Silaghi *et al.* 2000g) can be used with no modification for versions of ABTR where each agent always enforces all the constraints it knows (e.g. ABTR reordering technique for AAS). Alternatively, for use with the general version of ABTR, the solution detection technique presented in (Silaghi *et al.* 2000g) is modified by attaching as additional parameter of **accepted**(v) messages the *Set of References to the Constraints Completely Satisfied* by the partial valuation v, SRCCS(v). Each agent composing incoming partial valuations makes and attaches the union of the corresponding SRCCS.

The references in SRCCS refer only to initial constraints and not to constraints (nogoods) inferred during search. This algorithm assumes that agents know or may get references to (all) the constraints and that agents knowing the same constraint agree on a common reference for it. This is acceptable when the constraints are public. Otherwise, the technique in (Silaghi *et al.* 2000g) is applicable.

Proposition 20.2 *When the partial valuation V computed by the system agent by composing the last incoming valuations v_k from each branch k of the root of the solution detection spanning tree is not empty and SRCCS(V)=$\cup_k$(SRCCS(v_k)) contains references for all the constraints of the agents, then V is a solution.*

Proof. By construction, any extension of a partial valuation v satisfies all the constraints referred in the associated SRCCS(v). Therefore V satisfies SRCCS(V), satisfying all agents when SRCCS(V) contains references to all the constraints. □

Proposition 20.3 *The solution detection algorithm based on SRCCSs detects a solution for ABTR in finite time if the search does not fail.*

Proof. Let us assume that a solution was not detected before quiescence. Then according to Theorem 10.2 quiescence is reached by ABTR in finite time. Since at quiescence any initial constraint C is enforced by some agent, its reference is sent with the last valuation in the SRCCS along the spanning tree. Since at quiescence for ABTR the instantiations define a solution and are coherent, the constraint references propagate up to the root and all references are present in the last SRCCS(V) computed at root. □

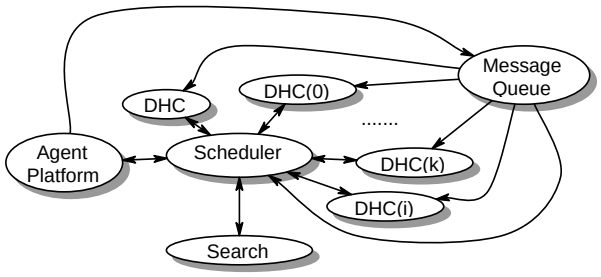

Figure 20.1: *The architecture of an agent A^i: active objects (threads) deal with each level of DMAC. In the idle time, the Scheduler thread performs backtrack search.*

20.3 Network Congestion Control in MAS

The problem of network congestion where messages made redundant can be discarded when they come from a single source have made the research subject of several communities (Vivas 2002). (Vivas 2002) has solved the problem attaching a bitmap of redundant messages to each message. From this point of view, our problem is even more general than that, since messages from one agent may have been invalidated by messages from other agents. On the other side, the definition of redundancy is much more simple and allows for important improvements.

Several incoming messages can contribute to the same level of consistency. Treating them individually, combined with the fact that each agent strives to do as much as it can, would result in uselessly fragmenting the local consistency output. Furthermore, since the time needed to send and process a messages is non zero agent buffers would risk to run short of memory. For this reason, our agent architecture (see 20.1) offers the opportunity to temper the flood of messages by compacting incoming messages. By this means, several messages of given types can be answered in block. This way, not only that the total number of messages is reduced, but also the space in local incoming buffers is spared, since all the messages are compacted in the receivers. For **propagate**, and **ok?** messages there is a common compactor for each level k in the search tree. The compactor for the level k is said to have the ID k, and is denoted DHC(k). The **nogood** messages are received by the compactor with the same ID as the current position of the receiving agent.[3]

The proposed "flow control" policy uses thresholds t, t^1, and t^2, and is defined by the following rules.

Rule 18 *The server refuses connections as long as the total size of the incoming queues grows over a certain threshold t^2.*

Connections are refused during a limited time since the size of the input queues necessarily decrease due to this blocking. To ensure that agents use this opportunity to make their computations without a memory overflow of the outgoing buffers, the next rule can be followed.

Rule 19 *When the total size of the outgoing queues increases over t, the output of the compactors towards the scheduler is blocked.*

This reduces the total number of messages in the network. Let VS be the maximal size of compacted information (v assignments and nv labels).

Proposition 20.4 *The flow control policy described here ensures that agents can act with a bounded space for communication buffers and without deadlock.*

Proof. The fact that the local space can be bounded is ensured by the Rule 18. Additional care is required for ensuring that the agents use the opportunity offered by Rule 18. Actually, in the presented schema, Rules 19 and 18 cooperate to maintain the size of the two queues. The agent can delay internal computations to generate new messages only when the output queue is emptied, such that the total space is bounded to $t^2 + t + S$, where S is the space required by the local algorithm.

[3]There is also one separate compactor for **accepted** messages.

Deadlocks cannot appear due to this flow control: Assume that a deadlock appears. It has to be related to some servers that are locked and do not accept messages. The compactors continue to compact the input queues until they reduce their space below t^2 or VS. Therefore, eventually, all input queues are reduced below t^2 and the servers are activated. □

Since overwhelming "avalanche" of messages appears seldom and temporarily in search processes, the proposed "flow control" is needed and induces no significant overhead. As presented, the compactors give priority to lower levels of important information. When an agent keeps receiving messages at a high rate that overcomes the computation capabilities of the agent, it is a good heuristic to first listen and compact the newer proposals than to hurry answering old ones.

20.4 Summary

In this last chapter we have mentioned some techniques that we had to develop for our implementations, even if they are only marginal to our interest. These techniques and the related problems will however interest all those that want to implement real distributed systems.

The termination algorithm described here is very similar to ones well known in literature. Our version is only slightly different as it is tuned to fit with the consistency achievement stages that appear in sMDC. We do not have knowledge of other solution detection algorithms, other than the ones described here. We have developed these ones as a natural alternative to the use of general termination detection algorithms for the asynchronous backtracking search. We have ended with a description of the flow control technique that ensures that our agents can communicate with a bounded space.