Distributed Constraint Satisfaction applied to Log-based Reconciliation

Yek Loong Chong[1] and Youssef Hamadi[2]

[1] University of Cambridge, Computer Laboratory, 15 J J Thomson Avenue
Cambridge CB3 0FD, United Kingdom, ylc21@hermes.cam.ac.uk
[2] Microsoft Research Ltd., 7 J J Thomson Avenue
Cambridge CB3 0FB, United Kingdom, youssefh@microsoft.com

Abstract. This paper presents four distributed algorithms for log-based reconciliation. Our methods remove the classical drawbacks of centralized systems like single point of failure, performance bottleneck and loss of autonomy. They are tolerant to failures and allow operations to perform multi-object updates without forcing full replication. The problem is formalized using the Distributed Constraint Satisfaction paradigm (DisCSP). In the worst case, the message passing complexity of our methods range from $O(p^2)$ to $O(2^p)$ in a system of p nodes. Experimental results confirm our theoretical analysis and allow us to establish quality and efficiency trade-off for each method.

Keywords: Distributed Constraint Satisfaction

1 Introduction

Optimistic replication is the enabling technology for Computer Supported Cooperative Work where users collaborate asynchronously to achieve a common goal. The major assumption in optimistic replication systems is that conflicting operations are exceptional [KR81] and can be resolved through a process called reconciliation. In these systems, users share a set of common data objects through replication and update their local replicas independently. These isolated updates can introduce divergence in the states of the shared objects which are occasionally repaired through reconciliation.

In log-based reconciliation, local updates are recorded in logs containing the isolated operations. These logs represent the input of the reconciliation process. The output or *schedule*, which is an ordered collection of non-conflicting operations, is computed by combining the isolated operations in some order. The schedule when executed against a previously consistent state results in a new consistent state. However, a set of isolated updates may contain conflicting operations and combining these operations may violate some invariant. One way of preserving correctness is to exclude some conflicting operations from the schedule. An efficient reconciliation algorithm minimizes these removals.

A major criticism of current optimistic systems is that reconciliation is centralized which exhibits classical drawbacks like single point of failure, performance bottlenecks and loss of autonomy. Drawbacks aside, due to various factors like organizational boundaries and security/privacy, centralization may indeed be impossible or undesirable. What is needed is a method for making decision locally and incrementally, without having to accumulate all information at one site. In this work, we present a set of distributed algorithms for log-based reconciliation which avoid the previous drawbacks and preserve privacy.

As demonstrated in [KRSD01], systems invariants can be represented using constraints. Such systems can then be modelled and solved in the Constraints formalism [Fag01, Ham04]. This approach presents several advantages like the insurance of finding the best possible schedule (optimal solution) and the clear distinction between modelling and resolution which brings easy system integration and seamless switch between solvers.

This work takes advantage of the previous contributions to provide the first distributed algorithms for reconciliation. These new algorithms are defined in the Distributed Constraint Satisfaction Problem paradigm (DisCSP) [Yok01]. A DisCSP is an abstract representation of a problem that is distributed amongst a group of autonomous agents cooperating to find a global solution. Each agent has a partial view of the global problem and is responsible for finding a solution to its share of the problem. Each agent is restricted by some constraints which must not be violated when searching for a local solution. Agents are also mutually restricted through shared constraints. A global solution is reached when every agent finds a local solution without violating both local and shared constraints.

This paper is organised as follows. We first provide a high level overview of the different domains in section 2. Section 3 presents the modelling of log-based reconciliation as a distributed constraint satisfaction problem. Our algorithms are presented in section 4 their theoretical analysis is given in section 5. Before the general conclusion provided in the last section, section 6 presents detailed experimental results.

2 Background

2.1 Constraint-based Optimistic Replication

In order to maintain consistency in a log-based reconciliation system, different updates to the same object must be executed in the same sequence at every replica. This can be complicated due to the varying latencies involved in accessing replicas that are distributed across a wide-area network. This problem is further exacerbated in mobile computing where the availability of replicas is also variable. Many optimistic systems like Bayou [DPS+94], IceCube [KRSD01], circumvent this problem by using a centralised commit protocol (also known as the primary-based commit). In these systems, a distinguished site or *primary* is elected by the system for committing tentative operations. The order in which tentative operations are committed at the primary is deemed to be the authori-

tative order. Tentative operations must be executed at all replicas in the order specified by the authoritative sequence.

IceCube is an optimistic replication system that supports multiple applications and data types using a concept called constraints between operations. The input is made by a set of logs coming from p users or sites $\{(a_{i1}, ..., a_{ij}) | 1 \leq i \leq p\}$ where a_{ij} represents the j^{th} action for user i. These logs are individually consistent i.e., operations recorded in a log do not violate any correctness criteria at the originating site. The schedule is an ordered collection of non-conflicting operations $(a_1, ..., a_k)$ which represents the largest subset of users' operations consistent with the following conditions:

1. "before" constraints (also called temporal): for any a_i, a_j in the schedule such that $a_i \rightarrow a_j$, a_i comes before a_j (not necessarily immediately before).
2. "must have" constraints (also called dependencies constraints): for any a_i in the schedule, $a_i \rhd a_j$ means that a_j must also be in the schedule.

Operations which are not part of the final schedule are deemed conflicting and aborted by the system. The semantics of a large set of applications can be represented through the combination of these low level constraints. For instance if the method f_m of an object o needs to start with the call to an initialisation method f_i we can use the constraints $o.f_i \rightarrow o.f_m, o.f_m \rhd o.f_i$.

A consistent schedule must respect any dependency relation and avoid oriented cycles of temporal constraints. The previous problem limited to *before* constraints can be reduced to the search of the largest acyclic network of actions. This problem is NP-hard [Kar72]. However, the addition of must have dependencies globally reduces the complexity. The reduction is even larger when the relations are symmetricals ($a_i \rhd a_j \rhd a_i$). Indeed, the previous introduces equivalence classes between actions which is similar to a reduction in the size of the problem [Fag01].

2.2 Distributed Constraint Satisfaction

DisCSP is a general paradigm able to represent various distributed problems. It is usually deployed in multi-agent applications where the global outcome depends on the joint-decisions of autonomous agents. Examples of such applications are distributed sensor networks management [FM02] and distributed planning [AD97].

Distributed optimization problems represent a new application field for DisCSPs. Recently, *ADOPT* the first distributed branch and bound algorithm was presented [PMY03]. This algorithm can find an optimal solution or a solution within a user-specified distance from the optimal. Of course, this was a first candidate for distributed log-based reconciliation. However an in-depth review concluded that ADOPT is too space intensive for our purpose. Indeed, assuming there are n nodes and each node is assigned one variable. The algorithm needs to maintain a context field which can be thought of as a partial solution at a specific node. At worst, the size of the context field at a node is n (i.e. the size

of the global solution). At node i, for each possible value that variable X_i can take i.e., D_i, the context of all its children must be recorded. Since a node has at most $n-1$ children, the space required to store the context field at node i is $O\left(|D_{x_i}| \times n^2\right)$. Although space requirement is polynomial, this algorithm is not suited for our needs. Indeed, our system has to manage large sub-problems distributed amongst several primaries. Therefore, a realistic approach is to perform local branch and bound searches combined with distributed satisfaction to correctly combine optimal local solutions.

3 Modelling

This section presents our distributed constraint modelling for the problem of log-based reconciliation. Considering that there are n tentative operations from p primaries, we use two constrained variables x_i and s_i to represent each tentative operation a_i. Therefore, a distributed reconciliation problem formulated as a DisCSP is a quadruplet (X,D,C,A) where:

1. X is a set of $2n$ variables $x_1, \ldots, x_n, s_1, \ldots, s_n$.
2. D is the set of domains $D_{x_1}, \ldots, D_{x_n}, D_{s_1}, \ldots, D_{s_n}$ of possible values for the variables $x_1, \ldots, x_n, s_1, \ldots, s_n$ respectively, where:
   - $D_{x_i} = [0..1]$, the value 1 is set when the action (or operation) a_i is selected for inclusion in the schedule.
   - $D_{s_i} = [-n^2..n^2]$, each value represents a possible position for action a_i in the schedule[3].
3. C is the set of predicates defining the constraints (see section 2.1) between the operations:
   (a) to express any $a_i \rightarrow a_j$ constraint, we use the logical invariant: $(x_i == 1) \Longrightarrow (s_i < s_j)$[4].
   (b) any dependency relation $a_i \rhd a_j$ is held by two logical invariants:
      i. $(x_i == 1) \Longrightarrow (x_j = 1)$
      ii. $(x_j == 0) \Longrightarrow (x_i = 0)$

   The first invariant expresses the fact that the inclusion of operation a_i cannot occur without the inclusion of operation a_j. The second invariant is redundant, it improves the efficiency of the resolution process (see [Ham04]).
4. A is a partition of X amongst the set of p autonomous primaries $P_1, \ldots P_p$ defined by $f : \mathsf{X} \rightarrow \mathsf{A}$ such that $f(x_i) = P_k \Leftrightarrow f(s_i) = P_k$.

[3] Intuitively, n discrete values would be sufficient to represent the positions of n operations. However, this is not the case with our distributed algorithms. Please see section 4 for details.

[4] It does not consider the possible exclusion of operation a_j. Since when $x_j == 0$, the entailment of $s_i < s_j$ is still consistent (see [Ham04]).

3.1 Local Optimization

As observed in section 2.2, distributed optimization is not realistic (so far) on real problems. Then our remaining choice is to combine local solutions in order to reach global consistency for the network of primaries. Each primary will only consider optimal solutions to its local problem. Here, the rationale is to keep the largest number of non-conflicting operations. We define, $(\sum_{\forall a_i \in P_k} x_i)$ as the objective function. This function is maximized by each P_k.

The previous search process is performed (using some Constraints solver e.g., [Ham03]) over the x_i with bound consistency on the s_i. At the end of the distributed exploration, any selected operation a_i (s.t., $x_i = 1$) can use the lower bound of the s_i as a valid ordering position in the schedule.

4 Distributed Algorithms for Log-based Reconciliation

4.1 Distributed ordering of the primaries

Our solution synthesis algorithms are totally asynchronous. However, they require an ordering between related primaries (two primaries are related if they share some constraint). Then, before looking for a distributed solution, we need to compute a partial ordering $>_o$ of the primaries. A general method, Distributed Agent Ordering (DisAO) is classicaly used in DisCSP to order a constrained network of autonomous agents [HBQ98]. This method can use any domain-dependent heuristic to guide the ordering. It computes a distributed acyclic graph of agents. This ordering is then used to support distributed search algorithms.

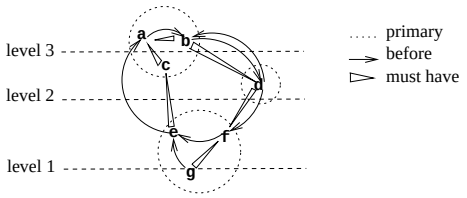

Fig. 1. Distributed Ordering of the Primaries

The figure 1 presents such an ordering. Each primary is presented with internal actions and constraints. Applying DisAO on this distributed reconciliation problem could result in any ordering between our three primaries. At the top level, the primary P_3 has no *Parent* but has two *Children*. The primary at level 2, P_2 has one parent and one child while at the bottom, the primary has just two parents. In the example, the resulting order is $P_3 >_o P_2 >_o P_1$. These *Parents* and *Children* sets represent primaries with higher (resp. lower) priorities. They come as inputs to our distributed search algorithms.

4.2 Asynchronous backtracking

We are now presenting in figure 1 our general distributed backtracking algorithm. It assumes that the nodes have been prioritized into some partial order.

For each primary, the input is made of the previously defined *Parents* and *Children* sets. The output is made by a local schedule consistent with both intra- and inter-constraints. Moreover this schedule is locally optimal (see section 3.1). When no local solution can be found, an empty schedule is returned. Such a schedule is considered as consistent. For space consideration, we do not comment the algorithm here. Interested reader can report to [CH04] for a full description and more experimental resutls.

Functions The algorithm is using a set of simple primitives/functions:

- *send(type, content, dest)*. Sends a message of some *type* (or id) which include the payload *content* to agents in *dest*. With respect to the problem's topology, this call only addresses "related" content.
- *resolveDomain(localView)*. Here we have to remember that the position of some operation a_i is stored in the related variable s_i (see section 3). Now, consider a primary P_k with n_k local operations. Intuitively, we only require n_k discrete values to represent the position of these operations. However, P_k searching for a local solution must also take into account of its *Parents* solutions. Clearly, some local operation a_i could be scheduled either between, before or after ancestors' operations. Then each domain D_{s_i} must locally be extended to accomodate all three cases.
  The *resolveDomain* function performs this "rewriting" of the domains. It is applied each time a new local solution $s' = ((x_{k'1}, s_{k'1}), ..., (x_{k'k'}, s_{k'k'}))$ (with $k' \leq n_{k'}$, the size of P_k's local problem) is received from $P_{k'}$.
  The resolution is a two step procedure. It first rewrites s' into s'' to make it consistent with the following invariant: $\forall s_{k'i}, s_{k'j} \in s', s.t.\ i \neq j, |s_{k'i} - s_{k'j}| > n_k$. Second, defines the new lower bound (resp. upper bound) of the local D_{s_i} domains as the $min(s'') - n_k$ (resp. $max(s'') + n_k$) where $min(s'')$ (resp. max) represents the min (resp. max) of the $s_{k'i}$ in s''.
  Now the new domains contain enough space to consistently solve the local problem with respect to current *Parents* states.
- *compatible(s)*. This function checks the compatibility of the solution s with the current local view. Basically, it checks inter-constraints' consistencies. If some inconsistency is found, the next search will be performed with respect to the local view and stored nogoods. It uses the previously extended domains D_{s_i}.
- *computeNogoods(conflicts)*. This function resolves nogoods explaining the current inconsistency. It returns a *conflicts* set which holds the sub-set of *Parents* involved in the inconsistency. It also returns a sub-set of the parents' actions involved in the conflict.
- *storeNogoods(ng)*, takes a set of nogood as argument and add them to the current set of constraints.

Algorithm 1: Distributed Asynchronous Backtracking

```
Input: Parents, Children sets;
Output: A local schedule consistent with intra- and inter-constraints;
begin
    tStamp:=0;
    localView:=∅;
    s:=maximize(local problem);
    send('infoVal', s + tStamp, Children);
    end:=false;
    while (!end) do
        msg:=getMsg();
        if (msg.type = "terminate") then end:=true;
        if (msg.type = "infoVal") then
            localView.add(msg.content); resolveDomains(localView);
            filterNogoods(msg.content);
            if (!localView.compatible(s)) then
                s:=maximize(local problem); tStamp++;
                if (s) then
                    send('infoVal', s + tStamp, Children);
                else
                    conflicts:=∅; ng:=computeNogoods(conflicts);
                    times:=localView.getStamps(conflicts);
                    send('backtrack',ng + times,conflicts);
                    localView.remove(msg.content);
                    s:=maximize(local problem); tStamp++;
                    send('infoVal', s + tStamp, Children);
        if (msg.type = "backtrack") then
            if (consistent(msg.content, tStamp)) then
                storeNogoods(msg.content);
                s:=maximize(local problem); tStamp++;
                if (s) then
                    send('infoVal', s + tStamp, Children);
                else
                    conflicts:=∅; ng:=computeNogoods(conflicts);
                    times:=localView.getStamps(conflicts);
                    send('backtrack',ng + times,conflicts);
end
```

– *filterNogoods(ng)*, removes the nogood set *ng* from the current set of constraints.

4.3 Example

To fully illustrate the algorithm let us go back to the example of figure 1. We have from top to bottom, primaries P_3, P_2 and P_1. Figure 2 provides from top left to bottom right a step by step execution of the algorithm. Each snapshot includes the local state of each action a_i in the form (x_i, s_i).

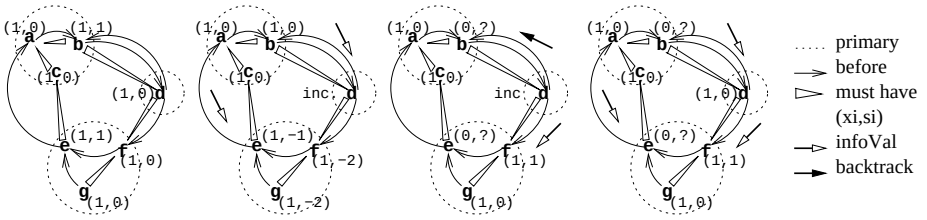

Fig. 2. A complete example of distributed backtracking

Processing starts on top left with the concurrent computation of a locally optimal solution for each sub-problem. At that step, all the actions are included in the schedule with respect to intra-constraints.

Then in the second snapshot, we assume that P_2 and P_1 are processing incoming *infoVal* messages upcoming from P_3. P_1 finds a new local solution consistent with P_3, while P_2 detects an inconsistency. This inconsistency comes from the cycle of *before* constraints between actions b and d combined to the dependency constraints $b \rhd d$. Indeed, removing action d in order to avoid the inconsistency upcoming from the cycle is not compatible with the dependency constraint. In the third snapshot, the nogood $(x_b \neq 1)$ is addressed to P_3 in a *backtrack* message. At the same time, P_1 processes the *infoVal* message upcoming from P_2. It finds a new local solution excluding e but compatible with both P_3 and P_2. In the last snapshot, P_3 finds a new solution which respects the stored nogood and propagates it towards P_1 and P_2. Then P_2 can find for the first time a local solution compatible with P_3. It addresses it towards P_1. Finally P_1 successively processes two *infoval* messages and finds them compatible with its current local solution. At that stage termination is detected. The overall quality is 5 while a centralized processing comes up with a quality of 6 by excluding action b. The overall message passing cost is 7.

4.4 Backtrack free search

The previous example shows that our algorithm can easily handle inconsistencies upcoming from distributed cycles of *before*. Indeed these cycles can always be

broken by removing some action located in the primary of lowest priority. For example, in figure 2 the cycle involving a, b, d, f, e can always be broken in P_1 by removing either f or e. Combinations of dependency constraints cannot force the system to backtrack. In fact, we can notice that backtracking occurs when some parent schedules some operation a which depends on some external operation b located in some child primary, i.e., $a \rhd b$. In such a situation, if b cannot be scheduled in the child, a has to be withdrawn. The previous configuration appears between actions b and d in figure 2.

From the previous observations we can easily derive a new algorithm. In this procedure, a node proactively exclude local actions whose scheduling depends on some child's decision. This new algorithm is backtrack-free (see section 5). When we apply this algorithm on the problem of figure 2, P_3 and P_2 proactively remove b and d. At the end of the resolution, the overall quality is 5 (P_1 can keep three actions) while the message passing cost is just 3.

4.5 Pre-processing

The last algorithm adopts a very aggressive strategy of coercing each node to exclude operations which rely on child's decisions. Even when these "dependent" operations can be successfully scheduled, they are needlessly excluded. These unnecessary exclusions can be reduced by prioritizing the primaries in an ordering which minimizes the number of actions that depend on child's decisions.

Figure 3 presents such an ordering applied to our initial problem (see figure 1). In this new ordering, action e relies on action c, which means that the top priority primary depends on the decision of a child in order to schedule e. This ordering represents one of the three optimal orderings. Indeed, an ordering that removes all the bad dependencies is impossible since the system presents an oriented cycle of *must have* constraints.

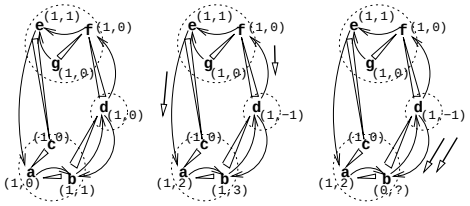

Fig. 3. Backtrack search with minimization of badly oriented dependency links

When we run our distributed backtracking on this ordering we come up with the optimal solution which schedules 6 actions (b's removal breaks the two distributed cycles of *before*). The overall message passing cost is 4 (see figure 3). Figure 4 applies the previously defined backtrack-free algorithm with the same ordering. It comes back with a schedule of quality 5 which proactively excludes e

and removes b in order to break a temporal cycle. At the end of the computation, an overall quality of 5 is achieved with 4 messages.

In section 4.1 we gave an high level overview of a distributed algorithm which computes a partial ordering of primaries by local propagation of information. This algorithm can be guided by any domain-dependent heuristic [HBQ98]. To apply the previous minimization process with DisAO we need to feed this algorithm with such an heuristic. We propose the following local behavior for each node P_i:

1. Compute locally inconsistent orderings, $P_j \rhd P_i \rhd P_k$.
2. Propagate these orderings in the neighbourhood.
3. Collect inconsistent orderings, prune local combinations of orderings by using collected information.

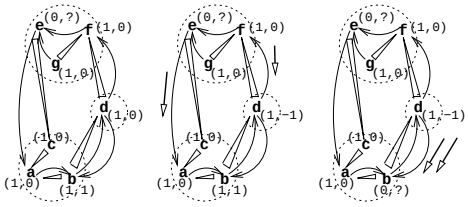

Fig. 4. Backtrack-free search with minimization of the badly oriented dependency links

The first step detects situations where some action in P_j relies on some action managed by P_i while at the same time some possibly different action in P_i relies on some action managed by P_k. These situations must be avoided.

The last step assumes that a node manages all the possible local orderings involving itself and its neighbours. It uses incoming inconsistencies to filter these local orderings. When it receives some inconsistency $P_j \rhd P_i \rhd P_k$, it prunes all the orderings assuming $P_j \rhd P_i$ or $P_i \rhd P_k$,

If we apply the previous heuristic on our initial example (figure 1), three ordering are excluded: $\{P_2 >_o P_1 >_o P_3, P_3 >_o P_2 >_o P_1, P_1 >_o P_3 >_o P_2\}$. At the end of DisAO, the three remaining orderings minimize the badly oriented dependency constraints (see figure 3 for such an ordering).

However, how far can we go with local decisions? It seems that in our situation, local reasoning can easily raise suboptimal orderings. In fact, the previous minimization process is equivalent to the computation of the smallest cycle-cut set on the inter-dependency subgraph.

Since inter-dependency constraints can be involved in several distributed cycles, an optimal solution will give the bad orientation to constraints representing the intersection of these cycles, i.e., to the smallest cycle-cut set. Figure 5 presents an example. On the left part we have a system with three pri-

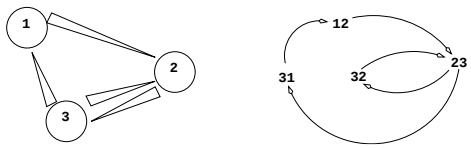

Fig. 5. Distributed cycles of dependencies and the associated dual graph

maries linked through two cycles of dependencies. An optimal ordering will decide $P_2 >_o P_3$. Indeed, this decision breaks all the cycles. The final ordering is $P_2 >_o P_1 >_o P_3$ (top to bottom).

To achieve this result we must work on the dual graph of the primaries. This is presented on the right part of the figure. In this graph, each node represents a dependency constraint of the initial graph, while each link connects dependency constraints with respect to the initial graph. This graph has two cycles. Computing the smallest cycle-cut set on the dual graph results in the identification of the node 23. The removal of this node breaks both cycles. When we go back to our problem, this is equivalent to deciding that primary 2 will have some action relying on some external action managed by P_3. This solution is optimal.

Computing the smallest cycle-cut set of a graph is an NP-hard problem which is equivalent to our initial problem limited to temporal relations. However several heuristics have been defined to tackle this important problem. Most of them are centralized, some are distributed. [JD97] has presented four distributed algorithms performing the distributed computation of a distributed cycle-cut set. The complexity of these algorithms range from $O(n^2)$ to $O(n^3)$ where n is the size of the graph. In our work, we wanted to evaluate the importance of a good or optimal ordering to tackle distributed log-based reconciliation problem. We decided to apply the previous cycle-cut set computation as a pre-processing step for DisAO. We first start with the computation of the smallest cycle-cut set of dependency constraints. We then put the cut set inter-constraints in the "bad" orientation. This decides $>_o$ for a subset of the primaries. After that step, we apply DisAO to order the remaining primaries. Now if we include the previous pre-processing step, we finally have four distributed algorithms:

1. The general distributed algorithm: *backtracking*.
2. The backtrack-free algorithm: *backtrack-free*.
3. The general distributed algorithm using the previous pre-processing: *pre-proc+backtracking*.
4. The backtrack-free algorithm using the previous pre-processing: *pre-proc+backtrack-free*.

5 Theoretical analysis

5.1 Distributed backtracking

Completeness The completeness of distributed backtracking is related to nogoods management. Filtering and discovery of nogoods are particularly critical. Our distributed algorithms use very simple but effective mechanisms to manage these nogoods. As said previously, nogood definition starts with some inconsistent inter-dependency constraint. Moreover, the detection of conflicting actions and primaries is restricted to the scope defined by the conflicting dependencies. Finally, when some *infoVal* information arrives, primaries have to filter their nogoods. We apply a very simple policy here which throw-away the whole set of local nogoods. The previous choices are easy to implement yet suboptimal.

Now, if we suppose incompleteness, some distributed solution is missed. Since each primary initially considers its whole local search space, the only possibility of missing a consistent state is to detect a nogood which subsumes it. Since nogoods are detected in inconsistent local states and deleted when local subspace must be entirely reconsidered, we reach a contradiction which proves the completeness of our distributed algorithms.

Correctness and termination Correctness is obvious since the state detection algorithm [CL85] stops the whole system when the communication network is empty and primaries are waiting for messages. At that ending point, each primary has come to a local state satisfying intra- and inter-constraints.

Acyclic orderings of primaries avoid situations where an agent gets back its search space through cascading *infoVal* initially started in some children. Indeed, since *infoVal* messages imply a local sub-space reinitialisation, cyclic orders may compromise the termination. Correctness, acyclic orders and the nogoods discovery and storage finally bring the exhaustion of the distributed search space. This proves the termination property.

Complexities Space complexity is polynomial. Time complexity consisders in the worst case, a fully connected network of p primaries. Each primary manages one action. The system exhibits all the possible combinations of inter-constraints. Any distributed ordering of this system has p levels. Insolubility is proven after an exhaustive exploration of the distributed search space. Since this exploration is done on the x_i variables, the search space size is 2^p (see section 3.1). Then in the worst case, time and message passing complexity are $O(2^p)$.

5.2 Backtrack-free search

Completeness, correctness and termination are obvious with this algorithm. However, the complexity must be explained. With the previous worst case, each primary proactively removes its local action to avoid incoming backtracking. The top level primary sends $p-1$ *infoVal* messages while the others send a number of message related to their position in the distributed ordering. Since this ordering has p levels, time and message passing complexities are $O(p^2)$.

6 Experimental results

Experiments were conducted over a set of randomly generated inputs. We simulated a system of $p = 10$ primaries with a total number of actions $n = 100$. Each primary P_k was allocated exactly $n_k = 10$ actions. Two tightness paramaters p_b and p_{mh} were used to define respectively the number of *before* and *must have* constraints. Each parameter represents a percentage over the set of all possible constraints.

Our motivation was to deeply evaluate the communication part of our distributed algorithms. Since using hard internal sub-problems impacts the overall quality but does not raise high interaction levels, we decided to use two formulations to define respectively the number of intra- and inter-constraints. The amount of intra-constraints is set to $tightness \times n_k(n_k - 1)$ with $p_b = 10\%, p_{mh} = 5\%$ as *tightness* parameters.

The amount of inter-constraints was define using $tightness \times (\sum_{k=1}^{p} n_k \times (n - n_k))$ with both tightness parameters ranging from 0.1% to 1.5%. These values ranges over the space of realistic log-based reconciliation problems. Indeed when the previous parameters are pushed they rapidly meet overconstrained regions with empty solutions (see [Fag01, Ham04] for full landscape analysis). With these formulations and our initial parameters, each primary had 9 intra-*before* combined with 5 intra-*must have*. Each system had from 18 to 270 inter-constraints either *before* or *must have*. For each combination of parameters we used 20 random instances. The connectivity of each instance was checked. Final values represent the average over the 20 instances.

As said initially, our approach is performing distributed satisfaction of locally optimal sub-problems. Distributed schedules were expected to be suboptimal. The top left part of figure 6 gives an overview of the quality loss with our *backtracking* algorithm on the whole parameters' space. To build this surface, we had to solve centrally each instance with a complete solver [Ham03]. The same solver is used within each primary to successively tackle local subproblems. The quality loss of the *backtracking* algorithm ranges from 0.003% at $p_b = 0.1\%, p_{mh} = 1.5\%$ to 36.7% with $p_b = 1\%, p_{mh} = 0.7\%$. More generally, our algorithm performs well in under- and over-constrained regions.

From top right to bottom right figure 6 presents for our four algorithms quality, message passing and efficiency results with $p_b = 0.5\%$, i.e., 45 inter-*before* constraints. When we consider the quality, we can see that *backtracking* and *pre-proc backtracking* are very close. It seems that an optimal or very good distributed ordering (which use the cycle-cut set calculation) cannot greatly improve the quality. On the other-hand, the *backtrack-free* method benefits from the pre-processing. The biggest gap appears at $p_{mh} = 0.2$ with an improvement of 14.5%. For this algorithm, the pre-processing organizes the system in a way which reduces the number of pessimistic choices used to avoid further backtracking operations. Message passing presents a different dynamic. The pre-processing clearly improves the *backtracking* algorithm. The biggest improvement appears with $p_{mh} = 0.2$ where 95% of the messages are saved. The *backtrack-free* does not experience message passing reduction. Indeed, as explained in 5.2, this algo-

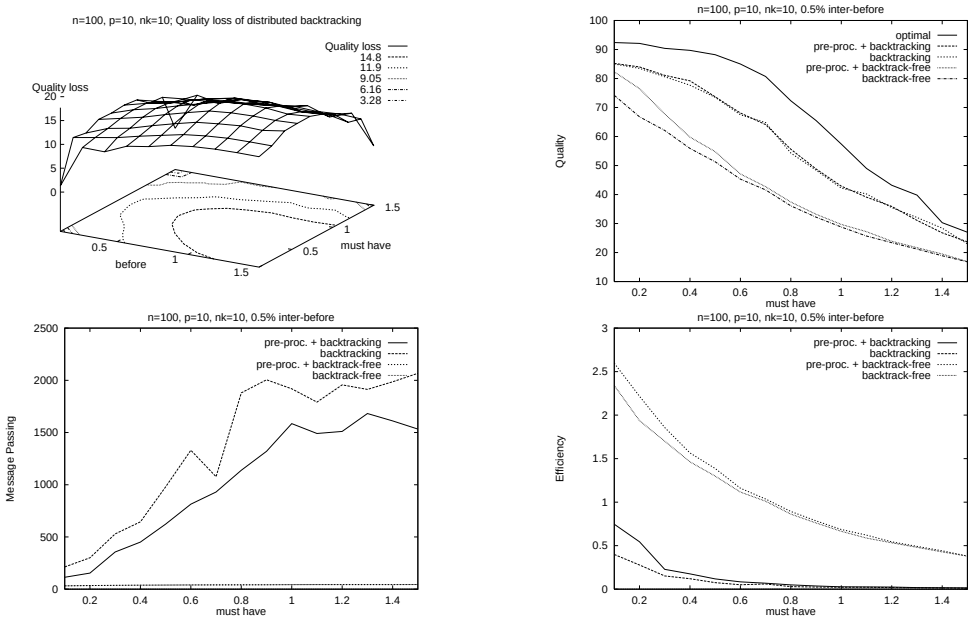

Fig. 6. Quality loss, quality, message passing and efficiency results

rithm performs top to bottom propagations of local solutions in the distributed system. These information are locally computed in a way which avoids further inconsistencies i.e., further backtracking messages. The message passing complexity remains the same. We defined the efficiency as the quotient between quality and message passing. This allows us to connect the final quality to the amount of communications required to reach it. We can see that for *backtracking* and *backtrack-free*, the pre-processing provides higher improvements in efficiency with under-constrained problems. The improvements drops with the addition of intra-dependency constraints.

7 Conclusion and future work

In this work, we proposed a set of distributed algorithms for log-based reconciliation. Our solutions are fully distributed and asynchronous. They avoid classical pitfalls associated to centralization. Furthermore, unlike a recently proposed decentralized protocol [Kel99], our solutions allow operations to perform multi-object updates without forcing full replication. Although the solution synthesis process requires a partial ordering of the nodes, the process is tolerant to failures. Should nodes become unavailable during the resolution, active nodes can still make progress since they always hold a consistent local solution. Experimental results confirmed our theoretical analysis of the algorithms. Moreover they allowed us to establish quality and efficiency trade-off for each method.

References

[AD97] A. Armstrong and E. Durfee. Dynamic prioritization of complex agents in distributed constraint satisfaction problems. In *Proc. of the 15th Int. Joint Conf. on AI (IJCAI-97)*, pages 620–625, 1997.

[CH04] Y. L. Chong and Y. Hamadi. Distributed algorithms for log-based reconciliation. Technical Report MSR-TR-2004-104, Microsoft Research, Cambridge, UK, December 2004. ftp://ftp.research.microsoft.com/pub/tr/TR-2004-104.pdf.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *TOCS*, 3(1):63–75, Feb 1985.

[DPS+94] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The bayou architecture: Support for data sharing among mobile users. In *Proc. IEEE W. on Mobile Computing Systems & Applications*, pages 2–7, 1994.

[Fag01] F. Fages. CLP versus LS on log-based reconciliation problems. In *6th ERCIM W. of the Constraint Group, Prague, Czech Rep.*, May 2001.

[FM02] S. Fitzpatrick and L. Meertens. Scalable, anytime constraint optimization through iterated, peer-to-peer interaction in sparsely-connected networks. In *Proc. Sixth Biennial World Conf. on Integrated Design & Process Technology (IDPT 2002)*, 2002.

[Ham03] Y. Hamadi. Disolver: A Distributed Constraint Solver. Technical Report 2003-91, Microsoft Research, Dec 2003.

[Ham04] Y. Hamadi. Cycle-cut decomposition and log-based reconciliation. In *In 14th Int. Conf. on Automated Planning&Scheduling, W. Connecting Planning Theory with Practice*, pages 30–35, 2004.

[HBQ98] Y. Hamadi, C. Bessière, and J. Quinqueton. Backtracking in distributed constraint networks. In *ECAI*, pages 219–223, Aug 1998.

[JD97] A. Jagota and R. Dechter. Simple distributed algorithms for the cycle cutset problem. In *Proc. of the 1997 ACM Symp. on Applied computing*, pages 366–373. ACM Press, 1997.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[Kel99] P. Keleher. Decentralized replicated-object protocols. In *18th Annual Symp. Principles of Distributed Computing*, Apr 1999.

[KR81] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6:213–226, 1981.

[KRSD01] A. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In ACM, editor, *Twentieth ACM Symp. on Principles of Distributed Computing PODC, Newport, RI USA*, 2001.

[PMY03] M. Tambe P. Modi, W.-M. Shen and M. Yokoo. An asynchronous complete method for distributed constraint optimization. In *The 2nd Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, 2003.

[Yok01] M. Yokoo. *Distributed Constraint Satisfaction:Foundation of Cooperation in Multi-agent Systems.* Springer, 2001.