PSolver: A distributed SAT solver framework
by
Daniel Kokotov
Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science and Engineering

at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2001

© Daniel Kokotov, MMI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.
PSolver: A distributed SAT solver framework

by

Daniel Kokotov

Submitted to the Department of Electrical Engineering and Computer Science
on August 26, 2001, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering

Abstract

In this thesis we present PSolver, a framework for distributed computation of the SAT problem. PSolver runs on general-purpose hardware and networks that support the TCP/IP protocol. It follows the voluntary computing platform model and exhibits a high degree of modularity, scalability and reliability. It builds on ideas from existing similar systems, in particular PSATO, while addressing the shortcomings of those systems, and incorporating several new ideas. We present the motivation behind PSolver, and describe its design and interesting aspects of its implementation. In addition, we discuss several important lessons about the design of distributed, multi-platform software systems we learned while working on PSolver. Finally we present some experimental results we've obtained from running PSolver on several benchmark problems.

Thesis Supervisor: Daniel Jackson
Title: Associate Professor
Acknowledgments

I’d first of all like to thank my thesis advisor, Daniel Jackson for his support of my work, helpful advice, and putting up with my outbursts of insolence and sloth.

I want to thank Ilya Shlyakhter for suggesting the original idea for this thesis, and helping immeasurably along the way.

I want to thank Allison, my officemate, for being fun and distracting me from working on this thesis. On second thought, maybe the latter is not a good thing. But it seemed good at the time.

I want to thank my roommates at the Dise for not throwing me out when i was late in paying my bills, because finishing this thesis while homeless would have been difficult.

I want to thank Chris Peikert for his helpful Latex advice, which made this document halfway pleasing to the eye.

I want to thank my parents for sending me money sometimes. Oh yeah, and also for their love, and support, and caring, and stuff.

I want to thank Sveta for telling me that my fears of not getting this thesis done were ungrounded, and my apprehensions of it lacking merit moronic.

I want to thank Marina for being the cool person that she is.

I want to thank Jessica for keeping me sane during the summer while i worked veverishly (a.k.a. slowly and sporadically) to get this done.

I want to thank the Cave Dwellers for being fantasy baseball studs.
Contents

1 Introduction 15

2 The SAT problem 19

3 The Standard Approaches 23
  3.1 Formulations of the SAT problem 23
  3.2 The Davis-Putnam procedure 25
  3.3 Overview of techniques and heuristics 27
    3.3.1 Unit Propagation 28
    3.3.2 Case Splitting 31
    3.3.3 Backtracking 32

4 Going distributed 35
  4.1 Overview 35
  4.2 The Bohm-Speckenmeyer parallel SAT solver 36
  4.3 The PSATO distributed SAT solver 38

5 PSolver: our approach 43

6 PSolver Design 51
  6.1 Architecture 51
    6.1.1 Network Topography 51
    6.1.2 The Participants 52
    6.1.3 Interprocess Communication 54
6.1.4 The PSolver Protocol ............................................. 54
6.1.5 Load Balancing .................................................. 54
6.2 High-level algorithms .............................................. 56
6.2.1 server ............................................................. 56
6.2.2 client ............................................................. 61
6.2.3 monitor ........................................................... 63
6.2.4 user ............................................................... 64
6.3 PSolver protocol .................................................... 65
6.4 Software design ..................................................... 67
6.4.1 The SendableObject Hierarchy - lightweight serialization ... 67
6.4.2 Message, MessageSet and MessageRegistry - parametrized mes-
saging ................................................................. 68
6.4.3 CommunicationObject - generic interface to message passing 71
6.4.4 Event/EventChecker/Callback/EventManager system - generic
parameterized event handling ............................................. 72

7 A step back: Why was it hard ................................. 83
7.1 Why distributed is hard ............................................. 83
7.1.1 Difficult to apply/visualize design patterns .................. 83
7.1.2 Type systems are difficult to implement ..................... 84
7.1.3 Memory management is difficult ............................. 85
7.1.4 Testing is difficult ............................................. 85
7.2 Platform independence ............................................. 85
7.2.1 Differences in implementation of same functionality ........ 85
7.2.2 Unavailable functionality ..................................... 86
7.2.3 Buggy portability tools ....................................... 86

8 Results ................................................................. 89

9 Future Directions ................................................... 93
9.1 Improving the PSolver protocol ................................. 93
9.2 Further use of STL ................................................. 94
9.3 Making more use of information from clients ...................... 94
9.4 Incorporating the progress bar ..................................... 95
9.5 A Java client ....................................................... 95
9.6 A web interface ..................................................... 95

10 Conclusion ......................................................... 97

A The PSolver protocol .............................................. 99
B Instrumenting Solvers ............................................. 111
C PSolver user guide .................................................. 115
  C.1 Setting up and running the system .............................. 116
  C.2 A Concrete Example ............................................ 118
  C.3 Command Reference ............................................ 120
    C.3.1 SATMONITOR ............................................. 120
    C.3.2 SATSERVER ............................................... 120
    C.3.3 SATCLIENT ............................................... 122
    C.3.4 SATUSER .................................................. 123
    C.3.5 COMMON CONFIGURATION FILE PARAMETERS ............ 124
    C.3.6 Using SATUSER .......................................... 125
List of Figures

3-1 Trie representation of a SAT instance .......................... 29
6-1 PSolver System Diagram .......................................... 53
6-2 PSolver: Interaction between server, user, and clients during solving of problem ............................................. 59
6-3 PSolver Monitor Interaction ....................................... 65
A-1 Psolver Protocol message grammar .............................. 99
List of Tables

8.1 Timings on common benchmarks ............................. 90
Chapter 1

Introduction

And in the beginning...

Take a bunch of variables, each of which can have one of two values: true and false. Throw in some NOT’s, OR’s, and AND’s. Mix it all together and you get a boolean formula. Now, the task is: figure out if there’s a way to assign true and false values to all the variables so that the whole formula is true, and if such an assignment exists, what is it. At first glance, this seems exactly the sort of problem of interest only to mathematicians, theoretical computer scientists and as a benchmark of computers’ processing power. Indeed, the SAT problem, as it is called, has been studied in depth in computability and complexity theory and a number of interesting theorems have been proven about it. Chief among these theoretical results is that the SAT problem was shown to be NP-complete, that is in the worst case impossible to solve for large instances in a reasonable amount of time.

However, it turns out that the SAT problem also has applications to a wide variety of more practical problem domains. These include model checking, planning problems, circuit design, AI, and others. Because of this, a lot of research has gone into devising efficient algorithms to solve particular instances of SAT problems. It turns out that there are algorithms that can solve many SAT instances arising from such real-world problems fairly quickly. In fact, often translating a problem in a domain such as circuit design to a SAT instance, solving the instance using one of these algorithms, and translating the results back into the original domain is often more
efficient that solving the problem directly using domain-specific techniques. Furthermore, it is more efficient both in the sense that the actual computation is faster, and more efficient in the sense that devising an algorithm to solve the particular problem is easier. This is because instead of having to come up with a complete algorithm for each kind of problem, one needs only to find a mapping from the problem domain into the SAT domain, which is often easier.

Our own interest in the SAT problem stems from the primary focus of the research done in our group. We’re mainly interested in building tools that help in the analysis and design of high-quality software. To that end, we have designed a language, Alloy, which is an object modeling notation in the spirit of Z and UML. It can be used to express models of systems at varying levels of abstraction, from very close to actual source code to very high-level problem descriptions[6]. Along with the language we have a tool, Alloy Analyzer, for finding various properties of systems described in the Alloy languages. Alloy Analyzer can check for consistency, find instances which fit the models, and check assertions about the models. It works by translating the constraints expressed in the models to boolean formulas, and then using SAT solvers to solve them[2].

Previously we had been using a suite of third-party SAT solvers, such as RELSAT, SATO, and SATZ, along with a layer which presented a common interface to the Alloy Analyzer tool and performed translation and solver management [2]. This works fine for smaller models which result in SAT instances taking a few minutes, or at most a couple of hours, to solve. However, we recently have been trying to model some larger systems which yield much larger SAT instances. Also, the original Alloy language made it difficult to build really big models. The next generation of the language, and the analyzer tool, will make it much easier to write large models. It is clear that such models will yield SAT instances taking days to solve on a single computer.

The obvious solution is to parallelize the SAT solving step and take advantage of the many often-idle workstations in our (and certainly any typical) research group. Unfortunately, it turns out that, although much effort and research has gone into designing sequential, single-threaded SAT solving algorithms, the parallel problem
has received comparatively little attention. The few parallel algorithms that do exist often require either special purpose hardware, special purpose networks, or are in other ways inconvenient to use.

Therefore, we decided to design our own distributed SAT solver that would address the shortcomings of the existing systems. Specifically, it would work on general-purpose computers connected by standard networks in no particular topology. Further, these computers would not be dedicated to running SAT solving algorithms - they would be computers belonging to people working on their own problems, that, to the extent that their full computing power was not being used, could donate some processing power to working on SAT problems. Most importantly, we wanted to deal only with the distributed aspects of the problem. It was clear that in any distributed SAT solving system, each of the individual computers would have to run some kind of a sequential SAT solver. We did not want to limit ourselves to any particular sequential algorithm, nor did we want to write our own. We were quite happy with the algorithms we were using in our original system (RELSAT, etc.) and wanted to be able to take advantage of any advances in sequential SAT algorithms. Thus a key requirement for our system was that it be able to work with any sequential SAT algorithm as its backbone.

The result of our work is PSolver, which we describe in the rest of this paper. It satisfies all of our original goals: it runs on general-purpose workstations connected by normal networks, it uses the voluntary computing paradigm, and any sequential SAT algorithm can be adapted to work with it. It is quite efficient and achieves near linear speedup for unsatisfiable SAT instances and sometimes super-linear speedup for satisfiable ones. We believe it to be the most complete distributed SAT solving system for general-purpose computers out there.

The rest of the paper is structured as follows. Chapter 2 gives some background on the SAT problem. Chapter 3 discusses some single-threaded algorithms and heuristics used to attack the problem. Chapter 4 presents other distributed and parallel systems for solving the SAT problem. Chapter 5 presents the goals and main ideas behind PSolver, our solution to the problem. Chapter 6 describes the design of the
system. Chapter 7 takes a step back and discusses the challenges we encountered and the lessons we’ve learned about designing distributed, multi-platform systems in general. Chapter 8 presents some experimental results obtained by running PSolver on some benchmark instances. Chapter 9 ponders future directions. Finally, chapter 10 concludes. In addition, in appendices A-C we give some technical details about the system. Appendix A gives the definition of the PSolver protocol - the set of messages used by the various pieces of PSolver for communication. Appendix B describes how to instrument an existing sequential solver so that it can be used with the PSolver system. Appendix C is a guide to setting up and running the system.
Chapter 2

The SAT problem

True or false?

Before the SAT problem was considered as a tool in solving practical problems, it received a lot of attention from theoretical computer scientists, playing a major role in the study of the P vs. NP problem. Recall that the P class of problems is the set of problems that can be solved in polynomial time on a Turing machine. The NP class of problems is the set of problems which can be solved in non-deterministic polynomial time. That is, we allow our Turing machine to make non-deterministic decisions in constant time, decisions such as “choose a to be either true or false”. If for every possible set of non-deterministic decisions the algorithm runs in polynomial time, then it is said that the algorithm runs in non-deterministic polynomial time. Another way to think about the NP class of problems is that it is the class of problems for which, given a candidate solution, there exists a deterministic polynomial algorithm to check whether the proposed solution is indeed a solution or not. It can be shown that the two definitions of NP are equivalent.

It is clear that the P class of problems is a subset of NP. What is not known is whether P and NP are equal or whether there are in fact NP problems which are not also in P. Complexity theorists have been unable to prove either that $P = NP$ or that $P \neq NP$, although most believe the latter is the case. The question is of vital importance and has huge implications for practical computer science. If a problem belongs to the P class, it means that it is generally practical to solve it directly for
fairly large instances. On the other hand, if a problem is not in $P$ - so no polynomial-
time algorithm exists to solve it - it is generally not possible to directly solve it for
reasonably large instances, and one must resort to approximation techniques. Many
important problems are known to be in $NP$ but are not known to be in $P$, i.e. no polynomial-time algorithms are known for them. If it were shown that $P = NP$, it
would mean that we would all of a sudden be able to efficiently and exactly solve a
wide variety of problems we had not been able to solve before.

It's easy to show that the SAT problem is in $NP$. Indeed, let us say that we
have $n$ variables connected by NOT's, AND's and OR's. The SAT problem is to
find an assignment for the $n$ variables such that the overall formula is true, or show
that no such assignment exists. Under the first definition of $NP$ (nondeterministic
polynomial time algorithm), we can give the following algorithm for solving SAT:

For each variable, non-deterministically choose it to be true or false. Then evaluate
the formula. If it is true for some choice of variables then we've got our assignment,
otherwise none exists.

Similarly, according to the second definition of $NP$ (a solution is verifiable in
polynomial time), given an assignment we simply check whether the formula is true
or false. It's fairly easy to see that we can evaluate a formula given an assignment in
polynomial time. For a more complete treatment, see Sipser's book [9].

Thus the SAT problem is in $NP$. No polynomial time algorithm is known for
solving it. But there's another important property that is known about SAT: it is
$NP$-complete. By that we mean that any other problem in $NP$ is as hard as it is
in the sense that any other problem in $NP$ can be efficiently reduced reduced to it.
More formally, a problem $A$ is $NP$-complete [9] if

1. $A$ is in $NP$

2. for any other problem $B$ in $NP$, there is a polynomial-time function $f$ that can
   map any instance of $B$ to an instance of $A$

$NP$-completeness is a very important notion, because essentially it singles out
some $NP$ problems as "representative" of the $NP$ class of problems. Thus we can
reduce our study of the $P \neq NP$ question to the study of the $NP$-complete problems. If one of them can be shown to be in $P$, then $P = NP$, because any other $NP$ problem can be converted to an $NP$-complete problem in polynomial time.

The SAT problem was the first to be shown to be $NP$-complete. There are now many problems known to be $NP$-complete, but for the most part the demonstration that they are $NP$-complete is done by showing that they can be converted in polynomial time to SAT - it is easy to see that this also shows they are $NP$ complete. SAT, on the other hand, was shown to be $NP$-complete from first principles. The proof can be found in any basic textbook on theory of computation.

Since most complexity theorists believe $P \neq NP$, it follows that they believe $NP$-complete problems are not in $P$ and hence cannot in the general case be solved on a computer within a reasonable amount of time. This is the case also for the SAT problem. It turns out however, that there are a variety of techniques that in practice make it possible to efficiently solve empirical SAT instances. In the next chapter we take a quick look at these techniques.
Chapter 3

The Standard Approaches

The olden days

There are several ways to represent the SAT problem and several ways to approach solving it in practice. In this chapter we explore the more common representations and solution techniques, focusing on sequential algorithms.

3.1 Formulations of the SAT problem

[4] To begin with, most SAT problems are given in the Conjunctive Normal Form (CNF). A SAT instance in the CNF form consists of:

1. A set of n variables: $x_1, x_2, \ldots, x_n$

2. A set of literals. A literal is a variable ($Q = x$) or the negation of a variable ($Q = \bar{x}$)

3. A set of m distinct clauses: $C_1, C_2, \ldots, C_m$. Each clause consists of literals combined by logical or (\lor) connectives.

4. The formula to be satisfied is the logical and of all the clauses: $C_1 \land C_2 \land \ldots \land C_m$

This can be seen as a special case of a Constraint Satisfaction Problem (CSP). The general constraint satisfaction problem consists of three components:
1. **n variables**: \(\{x_1, x_2, \ldots, x_n\}\). An assignment is a tuple of \(n\) values assigned to the \(n\) variables.

2. **n domains**: \(\{D_1, D_2, \ldots, D_n\}\). Domain \(D_i\) contains \(d\) possible values (or *labels*) that \(x_i\) can be assigned, i.e. \(D_i = \{l_{i,1}, l_{i,2}, \ldots, l_{i,d}\}\).

3. *A subset of* \(D_1 \times D_2 \times \ldots \times D_n\) *is a set of constraints.* A set of order-\(l\) constraints (\(l \leq n\)) imposed on a subset of variables \(\{x_{i_1}, x_{i_2}, \ldots, x_{i_l}\} \subseteq \{x_1, x_2, \ldots, x_n\}\) is denoted as \(C_{i_1, i_2, \ldots, i_l} \subseteq D_{i_1} \times D_{i_2} \times \ldots \times D_{i_l}\).

An order-\(l\) constraint indicates consistency/inconsistency among \(l\) variables for a given assignment. The variables conflict if their values do not satisfy the constraint.

We can see that a CNF-expressed SAT instance is an example of a CSP problem where each domain \(D_i\) is \(D_i = \{true, false\}\), and each clause with \(m\) variables corresponds to an order-\(m\) constraint.

Starting from the CNF expression, we can formulate the problem of solving the instance in several ways.

1. **Discrete Constrained Feasibility Formulations.** This is basically the most straightforward formulation - to find values for the variables given the constraints. The constraints can be expressed as a CNF formula, as above, or as a DNF formula (where the clauses are connected by \(\lor\)'s and each clause has literals connected by \(\land\)'s.

2. **Discrete Unconstrained Formulations.** These are essentially based on translating the CNF formula into numeric form and minimizing a function based on this translation. One common formulation is to minimize \(N(x)\), the number of unsatisfied clauses, under the interpretation that numeric variable \(x_i = 1(0) \iff \) boolean variable \(x_i = true(false)\). Then,

\[
\min_{x \in \{0,1\}^n} N(x) = \sum_{i=1}^{m} C_i(x) \quad \text{(3.1)}
\]
where

\[
C_i(x) = \prod_{j=1}^n Q_{i,j}(x_j) = \begin{cases} 
    x_j & \text{if } x_j \text{ in } C_i \\
    1 - x_j & \text{if } \bar{x}_j \text{ in } C_i \\
    1 & \text{otherwise}
\end{cases}
\]  

(3.2)

Similar formulations exist for DNF formulas.

3. **Discrete Constrained Formulations** Here we are again translating the CNF formula into numeric form but now placing additional constraints on the numeric function to be minimized. These additional constraints allow the search to escape local minima easier.

4. **Continuous Formulations** One can also interpret a SAT instance by translating the boolean variables to real-valued variables. This allows one to use various continuous-domain techniques and can “smooth out” infeasible solutions, leading to fewer local minima explored.

We will concern ourselves mostly with the first formulation: that is the discrete constrained feasibility formulation. We do this because it is the most basic one and the most-well studied one, and the one that forms the basis of our own system.

### 3.2 The Davis-Putnam procedure

[3] [10] The fundamental procedure for solving the SAT problem in this formulation is known as the Davis-Putnam procedure. The Davis-Putnam procedure is defined as follows:

```plaintext
function Satisfiable (set S) return boolean /* S is a set of clauses in CNF */
repeat /* unit propagation */
    for each unit clause (L) in S do
        /* unit-subsumption */
        delete any clause of the form (L or Q) from S
        /*unit-resolution */
```

25
for any clause of the form (not L or Q) delete not L from it

od

if S is the empty set return TRUE /* solution */
else if S contains an empty clause return FALSE /* conflict */
fi

until no further changes changes result

choose a literal L occurring in S /* case-splitting */
if (Satisfiable (S union {L}) return TRUE
else if Satisfiable (S union {not L}) return TRUE
else return FALSE
fi

end function

Thus the Davis-Putnam procedure is a series of alternating steps of case-splitting
and unit propagation, the latter in turn consisting of two parts - unit-subsumption and
unit-resolution. In unit propagation, we take a unit clause (i.e. a clause containing
just one literal), delete any clauses which contain that literal, since they are true
(this is unit-subsumption), and remove the negation of that literal from any clauses
which contain it, since it cannot make the clause true (unit-resolution). This may
create more unit clauses, so the step is repeated until no more unit clauses remain.
Then in the case-splitting step, we pick one of the unassigned variables, assign it the
value true (this is done by tacking on the unit clause L to S) and repeat. If the
unit propagation step results in S being the empty set this means we have found a
satisfying assignment. If the unit propagation step results in an empty clause, that
clause, and hence the entire CNF, is unsatisfiable under the current assignment and
we backtrack and try the value false. If that results in an unsatisfiable clause, we
backtrack further. Note that backtracking involves undoing any changes done in the
unit-propagation step - this is implied by the clause set being passed as the argument
of the function Satisfiable with pass-by-value semantics.
The execution of the Davis-Putnam procedure can be visualized as a binary tree, where each internal node corresponding a variable assigned a value by the case-splitting step in the algorithm, and each branch is labeled true or false and represents assigning that value to the variable for the node from which the branch originates. The left branch is the first value tried for each variable, and the right branch the second. Each leaf represents either a conflict, or a solution.

3.3 Overview of techniques and heuristics

[8] Given this basic procedure, we notice that there are three areas where there is room for figuring out heuristics to improve performance:

1. **Unit Propagation.** This is potentially a time-consuming step and is done many times. Efficient data structures for representing the clause set $S$ and algorithms for manipulating it can result in big savings over naive approaches.

2. **Case Splitting.** The decision here is which variable to assign a value to next, and which value to try first. A good choice will lead us to quickly finding either a satisfying assignment or a contradiction and so minimize the number of assignments, propagations and backtracks that need to be done. This is the crucial step in the algorithm that either makes or breaks it.

3. **Backtracking.** Sometimes, when a unit-propagation finds an empty clause and we need to backtrack, we can backtrack not to the most recently assigned variable but to an earlier one, based on certain information we have available - essentially information that tells us that the reason the partial assignment failed is not the most recent assignment but a previous one. This also saves a lot of times since it allows us to not even try certain assignments.

We now briefly mention some of the techniques that are used to improve performance in all three areas.
3.3.1 Unit Propagation

1. **Tries** This idea is used very effectively in SATO. The data structure and its use is described fully in the SATO paper; the following is a brief summary. [10]

Assume that each variable in the SAT instance has a unique index, which is a positive integer. The index of the negation of a variable is the negation of the index of that variable. A clause, then, is the list of the indices of the literals in the clause. The trie data structure is a tree whose edges are marked by indices of literals and whose leaves are marked by the clause mark. A clause is represented in a trie as a path from the root to a leaf such that the edges of the path are marked by the literal indices of the clause. To save space, if two clauses (represented as lists of integers) have the same prefix of length $n$, they should share a path of length $n$ in the trie.

If all the nodes that have an edge of the same mark to the same parent node in a trie are made into a linear list, we may represent the trie as follows: each node in the trie tree is either empty ($nil$), a clause end-mark ($\Box$), or a 4-tuple of the form $(var, pos, neg, rest)$, where $var$ is a variable index, $pos$ is the positive child node of this node, $neg$ is the negative child node, and $rest$ is its brother node. The interpretation is that the edge from this node to the $pos$ node is marked by $var$, the edge from this node to neg is marked by $-var$, and $rest$ is the “brother” node of this node - the next node in the linear list of nodes with the same parent nodes.

To translate a CNF formula $S$ into a trie $T_S$, we follow the same procedure:

(a) If $S$ is empty $T_S = nil$
(b) if $S$ contains an empty clause then $T_s = \Box$
(c) Otherwise, choose any variable index $v$ and divide $S$ into 3 groups:
   i. $P = \{v \lor P_1, \ldots, v \land P_n\}$, the clauses that contain $v$ positively
   ii. $Q = \{\bar{v} \land Q_1, \ldots, \bar{v} \land Q_n\}$, the clauses that contain $v$ negatively
   iii. $R = \{R_1, \ldots, R_n\}$, the clauses that do not contain $v$
Figure 3-1: Trie representation of a SAT instance

Let

i. $P' = \{P_1, \ldots, P_n\}$, $P$ with occurrences of $v$ removed

ii. $Q' = \{Q_1, \ldots, Q_n\}$, $Q$ with occurrences of $v$ removed

Let $T_P, T_Q, T_R$ be the trie nodes representing $P', Q', R$ (by applying this procedure recursively). Then $S$ can be represented by $T_S = (v, T_P, T_Q, T_R)$.

For example, let $S = (x_1 \lor x_3) \land (\bar{x}_2 \lor x_3) \land (\bar{x}_1 \lor x_2 \lor \bar{x}_3)$. Then $T_S = (1, (3, \Box, \text{nil}, \text{nil}), (2, (3, \text{nil}, \Box, \text{nil}), \text{nil}, \text{nil}), (2, \text{nil}, (3, \Box, \text{nil}, \text{nil}), \text{nil}))$, and the tree can be graphically represented as shown in picture 3-1.

Trie representation by itself does not guarantee efficient implementation of unit-propagation. In fact, trie-based implementations previous to SATO, such as DDPP and LDPP, were not very efficient. A key idea of SATO was to recognize that because backtracking occurs so often, the cost of undoing the effects of unit propagation can become significant. Furthermore, backtracking - which is the result of a unit resolution step that finds an empty clause - means that the
time spent on doing unit subsumption was wasted. Thus what is needed is a very low-cost way of doing unit propagation, in particular unit subsumption. It is even worthwhile to not do unit subsumption completely - so that sometimes the algorithm will consider clauses even if they had already been subsumed - if we can reduce the cost of unit propagation significantly, as long as eventually the fact that the clause has been subsumed is detected.

SATO accomplishes this by the following algorithm for unit propagation: it keeps a head list and a tail list of literals. The head list is the list of the first literals of each clause, while the tail list is the list of the last literals of each clause, the literals in each close being ordered by some measure (i.e. variable index). If the first literal of a clause becomes true it is removed from the head list. If it becomes false, it is removed from the head list and the next unassigned literal in the clause is added to the head list unless either (1) a literal with the value true is found in the clause, in which case the clause was already subsumed and no literal is added to the list; or (2) every literal in the clause has value false which means that the clause is empty and backtracking is required; or (c) the next unassigned literal is in the tail list, which means a unit clause has been found, in which case the clause is added to the list of unit clauses. The handling of the tail list is symmetrical.

It can be shown (see the SATO paper for a full analysis) that this algorithm performs unit propagation of assigning a value to variable $v$ in $O(N_v)$ if $v$ is assigned the value true and $O(P_v)$ if $v$ is assigned the value false, where $P_v$ is the number of clauses where $v$ appears positively and $N_v$ is the number of clauses where $v$ appears negatively. This is a significant improvement over a straightforward approach to unit propagation which would take $O((N_v + P_v) \cdot L)$, where $N_v$, $P_v$ are as above and $L$ is the average length of the clauses containing $v$. 30
3.3.2 Case Splitting

This is probably the most important aspect of a SAT algorithm, since the choice of the variable to assign a value to at each step affects how many total assignments need to be done, which ultimately decides the efficiency of the algorithm. Even with a very efficient algorithm for unit propagation, an algorithm which must make 100,000 variable assignments will run much longer than an algorithm that makes only 3,000 assignments even with inefficient unit propagation. There are several heuristics algorithms use to pick the variable, most of which boil down to one goal - choose the variable which, when instantiated, will result in the most constrained formula. Below we present some solvers' methods for selecting the next variable to assign a value to.

1. **The RELSAT algorithm.** [7] The RELSAT solver, which is the primary solver we use in both or Alloy tool and in PSolver, uses the following algorithm. It first checks if there are any binary clauses containing any unassigned variables. If there are none than a variable is chosen at random. If there are, then only binary clauses are considered in the rest of the algorithm. For each unassigned variable $v$ it then scores it as follows: $score(v) = 2 \times pos(v) \times neg(v) + pos(v) + neg(v) + 1$, where $pos(v)$ is the number of binary clauses containing $v$ positively, and $neg(v)$ is the number of binary clauses containing $v$ negatively. It then picks the (up to) 10 best candidate variables and re-scores them as follows. It runs unit-propagation on each candidate variable for both assigning true and false to the variable. If any such unit-propagation leads to contradiction, that variable is chosen immediately and assigned the value leading to the contradiction. Otherwise it re-scores each variable according to the same formula, but with $pos(v)$ being the number of variables assigned a value by running unit-propagate with $v = true$, and $neg(v)$ being the number of variables assigned a value by running unit-propagate with $v = false$. It then takes all variables within a certain “fudge factor” of the best score, and picks one of those at random.

2. **Bohm-Speckenmeyer Parallel SAT Solver algorithm.** The Parallel SAT
Solver by Bohm and Speckenmeyer is a parallelization of a sequential SAT Solver algorithm for a special-purpose message based MIMD transputer machine. In their sequential SAT Solver they use a lexicographic heuristic to pick the next variable to assign a value. For each unassigned variable $x$, they compute a vector $\{H_1(x), H_2(x), \ldots, H_n(x)\}$, where

$$H_i(x) = c_{\text{max}} \max(h_i(x), h_i(\overline{x})) + c_{\text{min}} \min(h_i(x), h_i(\overline{x}))$$

and $h_i(x)$ and $h_i(\overline{x})$ is the number of clauses of length $i$ containing $x$ positively and negatively, respectively, and $c_{\text{max}}$ and $c_{\text{min}}$ are empirically chosen constants. They then pick the variable $v$ with the maximal vector under the lexicographic order - i.e. the $H_1(x)$ are compared first, then the $H_2(x)$, etc. In their implementation they actually do a simplification of this algorithm and only consider the first two non-zero entries of the vector. They claim this does not significantly change the number of branching nodes.

### 3.3.3 Backtracking

The straightforward Putnam-Davis algorithm, when reaching a dead-end - a partial assignment which renders the CNF formula unsatisfiable - always backtrack to the most recent variable for which only one of the two values true and false has been tried, and tries assigning the other value to it. This, however, is often fruitless since the cause of the conflict - the partial assignment leading to unsatisfiability - may lie in an earlier-assigned variable. Therefore, some algorithms exploit techniques to prevent this from happening.

The most basic of these is clause recording, also known as nogood recording. The idea here is when a conflict occurs, to see which subset of the variable assignments cause the conflict, and to add a clause to the CNF which prevents that subset of assignments from being tried again. In addition to recording the clause, we also, for each variable, store not just the value currently assigned to it but also the depth of the search tree at which the assignment was made. Because recording clauses is both time
and space consuming, this process is often limited to keeping recorded clauses only of certain maximum size, and discarding larger clauses as soon as one of the literals in the clause becomes unassigned. This is known as \textit{k-bounded clause recording}, with \( k \) being the maximum size of clauses which are kept around indefinitely.

Note what this allows us to do. Now, when a dead-end is reached, we can jump back directly to the variable in the recorded clause with the largest depth, since we know any subsequently assigned variables could not have been responsible for the conflict.

In the RELSAT paper, Bayardo introduces an improvement on this technique he calls \textit{relevance-bounded learning}. The idea is basically to delete recorded clauses only when a certain number \( m \), of literals in the clause become unassigned, typically with \( m = 3 \) or \( 4 \). As a result, these recorded clauses may get used again in the future for either yielding conflicts or forcing variable assignments. This technique can be combined with \( k \)-bounded learning by keeping clauses of size less than or equal to \( k \) and deleting larger clauses only when \( m \) literals become unassigned.

Having now obtained a background in sequential SAT solving algorithms, we move on to the techniques for parallelizing them.
Chapter 4

Going distributed

The stiff competition

4.1 Overview

The number of parallel and distributed SAT solvers is far fewer than the number of sequential algorithms. This is somewhat surprising, considering that the SAT system would appear to be a perfect candidate for parallelization, both because it often involves extremely large problems and because it offers the potential for super-linear speedup from parallelization, in the case where a SAT instance is satisfiable.

The few parallel/distributed SAT solvers that do exist for the most part require special hardware. This includes solvers written for reconfigurable computing systems such as FPGA’s, solvers written for supercomputers and multiprocessors, and for various other special purpose machines such as transputers. These solvers rely on the special capabilities of the hardware for their parallelization. An example of such a system is the parallel SAT solver by Bohm and Speckenmeyer (called the B-S solver from now on) we mentioned in the previous chapter. We will now briefly describe this solver as representative of the special-hardware-dependent solvers.
4.2 The Bohm-Speckenmeyer parallel SAT solver

As mentioned before, the B-S solver runs on a 256-processor transputer system. A transputer system allows the programmer to realize any net-topology with the restriction that every processor is connected with at most 4 other processors. The configurations on which the B-S solver was run were 2-dimensional grids of up to $16 \times 16$ processors and linear processor arrays.

The B-S solver works by running a copy of their sequential solver (which basically implements the Davis-Putnam procedure with the case-splitting heuristic described in the last chapter) on each processor, and the $N$ processors cooperate during the search for a solution in the space of truth assignments by partitioning that space. The key part of any parallelization of a SAT solver is workload balancing, distributing the work between the participants so that the idle times of each participant without the workload balancing algorithm taking too much time.

The main idea behind the B-S workload balancing algorithm is to equalize the workload across processors. It does so by, at regular intervals, calculating the average workload and re-balancing the workload between the processors so that each processor's workload is equal to the average workload. The algorithm is as follows, in the case of a linear processor array:

1. Each node $p_i$ receives from its left-hand neighbor $p_{i-1}$ the prefix sum of the workloads $\tilde{\lambda}(p_{i-1}) = \lambda(p_1) + \cdots + \lambda(p_{i-1})$, where $\lambda(p_i)$ is the current workload of processor $p_i$. It then calculates its own prefix sum $\tilde{\lambda}(p_i) = \tilde{\lambda}(p_{i-1}) + \lambda(p_i)$ and passes it along to its right-hand neighbor.

2. $p_N$ calculates the optimal WL: $\mu = \frac{\tilde{\lambda}(p_N)}{N}$ and broadcasts $\mu$ to all processors.

3. Each processor calculates overload: $l(p_i) = \tilde{\lambda}(p_i) - i\mu$ and $l(p_{i-1}) = l(p_i) - (\lambda(p_i) - \mu)$.

4. If $l(p_i) > 0$ then send WL $l(p_i)$ to $p_{i+1}$ else receive WL $||l(p_i)||$ from $p_{i+1}$.
   
   If $l(p_{i-1}) > 0$ then receive WL $l(p_{i-1})$ from $p_{i-1}$ else send WL $||l(p_i - 1)||$ to $p_i - 1$.
Note that in steps 3, the overload or underload of the processor \( p_i \) is determined with respect to the total load of processors \( p_1, \ldots, p_{i-1} \). In step 4 a processor \( p_i \) may be required to send a workload greater than \( \lambda(p_i) \). This is okay; it will simply have to wait to receive sufficient workload before sending this.

The workload balancing algorithm for 2-dimensional grids is a simple extension of the linear array algorithm - it basically executes the linear array workload balancing algorithm for linear arrays in each dimension.

The above description leaves out two important specifics, namely how is the workload \( \lambda(p_i) \) of processor \( p_i \) calculated, and how are workloads transferred between processors. This is done as follows.

Given an overall CNF formula \( F \), let \( A \) be a partial truth assignment and \( F_A \) be the formula resulting from \( F \) by instantiating the variables in \( A \) and performing unit-propagation. Then the workload represented by \( F_A \) is \( WL = \lambda(F_A) = \alpha^n - ||A|| \) where \( n \) is the number of variables in \( F \) and \( \alpha \) is an experimentally determined constant which depends on the average clause length. It was determined by running their sequential solver on random formulas with varying clause length and plotting result times as a function of \( n \). They determined that this function was best approximated by \( \alpha^n \), with \( \alpha \) depending on the clause length. For example, for clause length 3, \( \alpha = 1.04 \)

Now, each processor maintains a list of sub-formulas to be solved. It works by taking a sub-formula \( F_A \) from the top of the list, and splitting it into two sub-formulas by picking a variable \( x \) and setting it to true and false. This creates two sub-formulas: \( F_{A_u(x)} \) and \( F_{A_u(x)} \). These are added to the bottom of the list. Periodically the workload-balancing step runs. It calculates the workload of processor \( p_i \) as \( \lambda(p_i) = \sum_{allF_A} \text{list of subformulas of } p_i \lambda(F_A) \). If in steps 3 and 4 it needs to share some amount of workload \( l(p_i) \), it does so by taking sub-formulas from the bottom of the list until their combined workload is as close to \( l(p_i) \) as possible without going over, and then sending them to the target processor. Since a sub-formula \( F_A \) is fully represented by the list of literals in \( A \), it is sufficient to just send those lists.

This is the algorithm in a nutshell. Our description leaves out several details and heuristics they use to simplify the algorithm in some cases. Refer to their paper for
a complete specification of the algorithm.

4.3 The PSATO distributed SAT solver

[5] There is one distributed SAT solver that we know of which works on general-purpose computers. This is the PSATO solver, based on the sequential SATO solver mentioned in the previous chapter. PSATO, though it has several faults, inspired our own work and provided us with some key ideas.

The key idea introduced by SATO is the use of guiding paths to keep track of the state of computation. Guiding paths actually came up in the context of solving the somewhat simpler problem of cumulating search. Given that solving a large SAT instance using a Davis-Putnam based algorithm can often take a very long time, sometimes it is desirable to split the computation into several parts. This may be because processor time is only available during certain times of the day, or it may be involuntary, caused by system crashes. Regardless of the cause, the goal is to be able to resume computation where it left off, without having to do work that has already been done.

This raises the question of what exactly is “work that has been done” in the context of solving a SAT instance, or more generally what is the search space of a SAT problem. There is no standard definition - it depends on the domain in which the SAT problem is interpreted, and on the algorithm used to solve it. But if we assume that the algorithm is based on the Davis-Putnam procedure, as described in chapter 3, then we can consider the search tree of Satisfiable for the given SAT problem as its search space. The search tree for a particular SAT problem depends on the splitting rule used by the particular algorithm. Once that rule is fixed, as well as any randomness in the algorithm, the search tree is fixed as well.

The algorithm visits the search tree by depth-first search. If in the process of solving a SAT problem the algorithm halts prematurely, it stops at some node in the search tree. As long as we have a way of encoding that node, we can then have the algorithm resume the search from that node and not look at any of the nodes already
explored.

A guiding path is just that: an encoding of a node in the search tree as well as (implicitly) of the nodes which have already been explored. The components needed to encode this information are:

1. The current partial assignment of the variables, as well as the order in which they were assigned

2. For each variable assigned, we also need to know whether the current assignment is the first value the algorithm has tried (and hence the other value must still be tried) or the second value the algorithm has tried (and so we do not need to try the other value).

A guiding path provides this information. Formally, it is an ordered list of pairs of the form:

\[(L_1, \delta_1), (L_2, \delta_1), \ldots, (L_k, \delta_k)\]  (4.1)

Here, for each pair, \(L_i\) is a literal representing an assignment to a variable. More specifically, it represents a recursive call to \(\text{Satisfiable}(S \cup L_i)\). Therefore if \(L_i\) is \(v_i\) for some variable \(v_i\), it represents assigning \(\text{true}\) to \(v_i\), whereas if \(L_i\) is \(\overline{v_i}\), it represents assigning \(\text{false}\) to \(v_i\). \(\delta_i\) is either 0 or 1. A value of 1 means that the other value for that variable still needs to be explored, and such a pair is called an open pair. A value of 0 means that the other value for that variable does not need to be tried, and such a pair is called closed. Thus, a guiding path pinpoints a node in the search tree as well as implicitly enumerates the nodes which have already been explored, and so is sufficient to describe the work-so-far of a Davis-Putnam based algorithm. All we need is to modify the Satisfiable procedure to accept a guiding path as input along with a set of clauses. The modification comes at the case-splitting step. If the guiding path is not empty, then instead of picking a variable using the normal case-splitting rule, we extract the first pair in the guiding path and use its literal to make the recursive call. If the pair is closed, then only that first recursive call is made; if it is open, then the second recursive call is made as well. This modified procedure is called Satisfiable-guided.
This is clearly a good solution to the cumulating search problem, but it is not immediately clear how to extend it to the domain of distributed SAT solving. The key lies in the ability to split a guiding path into two guiding paths so that running Satisfiable-guided with one of the guiding paths as input, then running it with the other guiding path is equivalent to running it with the original guiding path. Furthermore the two new guiding paths are such that they represent disjoint parts of the search tree; that is, the set of nodes visited by Satisfiable-guided with the first guiding path as input and the set of nodes visited by Satisfiable-guided with the second guided path is input is disjoint. This ability allows us to treat guiding paths as descriptions of subproblems, and generate a set of guiding paths which, combined, represents the entire original SAT problem.

The rule to split a guiding path in this manner is this: given a guiding path \( P \):

\[
P = (L_1, 0), \ldots, (L_i, 1), \ldots, (L_k, \delta_k)
\]

where \((L_i, 1)\) is the first open pair in \( P \), the two splits of \( P \) are \( P_1 \) and \( P_2 \):

\[
P_1 = (L_1, 0), \ldots, (L_i, 0)
\]

\[
P_2 = (L_1, 0), \ldots, (L_i, 0), \ldots, (L_k, \delta_k)
\]

It is clear, upon reflection, that this rule for splitting guiding paths satisfies the claim we made in the preceding paragraph: that the search space represented by \( P_1 \) and that represented by \( P_2 \) are disjoint, and their union is the search space represented by \( P \).

Armed with guiding paths as ways of representing SAT subproblems, we can now look at the architecture of the PSATO solver. It has a few key features:

1. It uses a master-slave model arranged in a “star” network topology. The master is responsible for keeping track of all the slaves, assigning them work, collecting their results and telling them to stop.

2. The master communicates a subproblem to a slave by sending it the pair \( S, P \),
where $S$ is a set of clauses corresponding to the original SAT instance and $P$ is a guiding path representing the particular portion of the search space the slave is to explore.

3. The slave solves the subproblem assigned to it by running Satisfiable guided until one of three events occurs: it finds a solution, it determines no solutions exist in the part of the search space it is assigned to explore, or it is ordered to halt. In the first two cases, the solution or the fact that none exists is reported back to the master. In the last case, the solver sends to the master a new guiding path representing the portion of its original subproblem it has not yet solved.

4. The master maintains a list of guiding paths. The number of paths in the list is kept at about 10

5. The list of paths is sorted by increasing length, and paths of equal length are sorted by increasing number of open pairs, with remaining ties broken arbitrarily. Whenever a slave becomes idle (after finishing a subproblem and finding no solutions) the master extracts a path from the top of the list and assigns it to that slave.

6. The algorithm terminates when either a solution is found, the clause set is determined to be unsatisfiable, or a specified period of time passes. In the latter case, the master tells each slave to halt, collects the paths returned by the slaves in response to this, and saves this set. On the next run this set can be used to resume computation where it left off.

7. One aspect of the system which is not readily apparent is how to “bootstrap” the system - given $n$ slaves, how to generate the initial $n$ paths to give them. In PSATO this is done by, in the beginning the entire clause set was assigned to one slave and it was allowed to run for 1 second. After that the resulting path was split into more paths and distributed among the other slaves which were allowed to run for 1 minute. After that the allowed time to run for each slave
Again, this description focuses on just the salient aspects of the PSATO system. For more details refer to their paper.

Now that we have gotten a solid grounding in the minutiae of the SAT problem and have explored some sequential and distributed techniques for solving it, we’re ready to move on to the meat of this paper: the arrival on the scene of PSolver, and how it left its competitors in the field of distributed SAT solvers in the dust.
Chapter 5

PSolver: our approach

We are the champions, my friend

After surveying the field of contenders for the title of best parallel/distributed SAT solver, we can see that they all have major flaws. The most significant shortcomings they exhibit are:

1. **Hardware Dependence.** Most of the existing parallel/distributed SAT solvers are written for a particular hardware architecture, whether FPGA arrays, multiprocessors, transputers, or other systems, and rely on the special features of that hardware. They are thus nearly impossible to port to other systems.

2. **Problem domain dependence.** Many of the solvers are designed to work with particular kinds of SAT instances. For example the Bohm-Speckenmeyer solver was optimized to solve specially generated random $k$-SAT instances, where all clauses are of equal length $k$ and are randomly generated - it uses a number of heuristics which depend on constants empirically calibrated for such instances. Other solvers are optimized for SAT instances arising from particular problem domains. While they will for the most part work with any SAT instances, their performance is sub-par compared with the types of instances they were optimized for.

3. **Sequential algorithm dependence.** Virtually all of the parallel/distributed solvers include their own implementation of the sequential SAT algorithm and
will only work with that algorithm. However, algorithms perform differently on different kinds of instances - one algorithm might do better than another on one kind of instance, but worse on a different kind. Also, improvements to the sequential solver algorithms are being made all the time and it seems unfortunate not to be able to take advantage of them.

4. **Poor software quality.** For whatever reason, many of the current solvers, both sequential and parallel/distributed, are poorly written from a software design point of view (the culprits will remain unnamed for fear of incurring wrath from their authors). This is not to say that they are inefficient or buggy. Rather, the problem is that the code is often poorly structured and scarcely, if at all, documented. This makes it very difficult to maintain and extend, especially for anyone but the original author. This is especially problematic for software written in a university research group, as ours is, where the makeup of the group changes very frequently.

These shortcomings convinced us of the necessity for a new system. Certainly if we want our Alloy tool to use a distributed SAT solver, hardware compatibility is a major issue and rules out all of the systems we looked at except for PSATO. The latter, though, suffers from a near total lack of documentation and support and even some known bugs. Thus, a new solver was called for and PSolver was born.

In addition to addressing the shortcomings of the existing systems, we had a few ideas in mind which would differentiate our system from the others. We settled on the following goals for PSolver.

1. **Modularity.** We saw the problem of building a distributed SAT solver as being composed of two fairly disjoint components. The first component encompasses the specifically distributed aspects of the system. These include workload balancing, communication between components, fault tolerance, persistence, and scalability. The second component involves the SAT solving aspects of the system - actually solving a given a SAT subproblem. One of our main goals was to cleanly separate these two components and to focus solely on the first - the
distributed aspects. As for the second component, we felt that the many sequential SAT solvers out there were good enough and we could just use them in our system.

2. **Extensibility.** This goal is closely related to modularity. We wanted all parts of our system to be easily extensible. This meant, for example, easily being able to adapt a new sequential SAT solver algorithm for use in our system, easily being able to change the heuristic used to do work balancing, easily being able to add new kinds of behavior to our system.

3. **General purpose hardware/networks.** We wanted to build a distributed SAT solver that ran on general-purpose workstations connected by standard kinds of networks. If our solver could also take advantage of multiprocessors, that would be an added bonus, but not a requirement.

4. **Hardware/OS independence.** Since we would not be targeting a specific hardware architecture, we needed a system that could easily be ported to different processor architectures and operating systems.

5. **Persistent SAT solving clusters.** This goal also originates from our target environment of networked workstations. We wanted to be able to create virtual clusters of machines devoted to solving SAT problems, and to be able to draw on these at any time. That is, we did not want to, every time we needed to solve a SAT problem, be required to manually run a client on every machine we wanted to use for our distributed system. This is quite inconvenient and would especially be cumbersome if the machines we wanted to use were located in different geographical areas and had different users. Rather, we wanted to set up a semi-permanent SAT solving network, so that whenever we had a SAT problem to solve, we could connect to it in some way and have it solve the problem for us.

6. **Scalability.** We wanted our SAT solving clusters to handle very large numbers of machines spread out over a potentially large area.
7. **Quality software design.** We wanted to write high quality, well documented code and incorporate the latest software design ideas, such as design patterns.

8. **Performance.** Last but not least, we wanted our system to offer good performance - close to linear speedup in the case of unsatisfiable SAT instances and super-linear speedup for satisfiable ones.

This is certainly a lofty list - pretty much we wanted to fix everything we thought was bad about existing systems and add a few extra features to boot. While we cannot claim to have achieved all of our goals to perfection, and we have not been using PSolver long enough to make conclusive judgements, we think that we have accomplished most of our objectives: modularity, extensibility and hardware independence, with good support for persistent clusters and scalability, while exhibiting fairly good performance. Below we briefly describe the way we achieved the goals stated above. The next few chapters will describe these, and PSolver in general, in more detail.

1. **Modularity.** This goal was achieved by architecting the system as consisting of four essentially different components collaborating on solving a SAT problem. These components are:

   (a) **server.** A server is the centerpiece of the system. It receives problems to solve from *users* and coordinates their solution. It keeps track of *clients*, splits up the main problem into subproblems, assigns the subproblems to clients to solve, collects the results and reports them to the *users*.

   (b) **client.** The client is the workhorse of the system. It is the piece that does the actual solving by running a sequential SAT solver on subproblems assigned to it. The client has the ability to locate servers by querying a *monitor*.

   (c) **monitor.** The monitor is the watchdog of the system. It keeps track of *servers*’ locations, by having servers register with it when they start running. The monitor then provides *users* and *clients* with server locations.
(d) **user.** The user is the interface to the system. Strictly speaking, the user is not a part of PSolver per se but in fact a user of PSolver, but for the purposes of discussing the PSolver architecture, it is useful to think of the user as a participant.

The key to this architecture is that these components communicate by passing messages drawn from a well-defined **PSolver protocol.** This is a fairly simple-text based protocol that fully defines possible interactions between components. Thus, in a sense the system is completely defined by the protocol and a specification which states the semantic meaning of each messages, the situations in which various messages can be sent and the responses different messages called for. The result is a system with very clean separation between components and well-defined interaction between them.

2. **Extensibility.** The PSolver protocol is also what makes the system very extensible. Because each component is essentially defined by the messages it can send and receive and when to send and receive them, one can easily write new components for the system by simply implementing this protocol. Thus, for example, new sequential SAT solvers can be incorporated into the system simply by writing a wrapper around it that is able to understand PSolver protocol messages, carry out their requests, and send reply messages. Similarly new server implementations that perhaps have different load balancing strategies can be done by implementing the protocol. Further, the protocol itself can be extended to define new kinds of messages. The system is also extensible on another level. We provide implementations for all four pieces, and our implementations are written in such a way as to allow one to easily modify key aspects of the four components. For example, the server is written so as to allow one to easily change the load balancing algorithm. The client is written to enable one to plug in different sequential solvers without the need to fully implement the protocol from scratch. We also provide an implementation for the monitor and a simple user interface which "implements" the user component.
3. **General purpose hardware/networks and OS independence.** This goal is satisfied almost by default. The PSolver protocol is a simple text-based protocol and so can be implemented on any machine. Components interact by passing messages, and so only require a network that supports an interface which allows for sending of text messages with some reasonable delivery semantics such as those provided by TCP/IP. In practice, our implementation is written in Standard C++ directly with message passing implemented directly on top of the sockets system interface. The few system-dependent calls we make are POSIX-compliant and so will almost certainly be supported on any UNIX-based system. On Windows systems, POSIX emulation is accomplished with the Cygwin package, which is under the GNU public license. All system-dependent calls are encapsulated in a subsystem which presents a higher-level system-independent interface to the rest of the code, so that porting, if required, should not be difficult. Thus far we have compiled and run our system on Solaris, Linux, and Windows workstations. We have had to port our sockets subsystem, initially implemented on top of the POSIX sockets library, to the Winsock socket library for the Windows distribution because of bugs in the Cygwin implementation of the POSIX library.

4. **Persistent SAT solving clusters.** This was accomplished by adopting the voluntary computing platform model pioneered by SETI@home and Shoutcast. The client component of the system is designed to run continuously, without intervention on the part of the operator of the machine on which it is running, and to be able to find servers and to deal appropriately with server crashes and temporary unavailability. The monitor component is what makes this possible. The monitor is a very simple, computationally inexpensive component that runs at a few well known locations and keeps track of currently running servers. Clients and users query one of these monitors when they initially start running and whenever the server they're currently connected to goes down. Thus anyone wishing to become a contributor to a SAT-solving cluster needs
simply to download our distribution and configure the client to run whenever the machine is booted and provide it with a list of monitor locations (presumably provided by the maintainer of the cluster). The client can be set up to use only a small percentage of the machine resources unless it is completely idle. The maintainers of the SAT-solving cluster then need to only run servers on a few machines and make sure to restart them if they crash. Anyone wishing to use the cluster needs simply to run a user and give it a list of monitor locations.

5. Scalability. The topology of PSolver (meaning of a PSolver-based SAT-solving cluster) is essentially what one might call a galaxy: several servers with many clients and some users connected to each server in a star configuration, and several monitors which servers register with and clients and users occasionally query. Furthermore, there can be multi-level server-client stars, since a client to a server can in turn actually be a server to its own clients. This provides for several places where a high level of scalability can be accomplished. The monitors route users and clients to different servers so that if the number of clients and users per server becomes to great, new servers can be started to alleviate the burden. Each server accommodates many clients and can be solving many problems from different users at the same time. And the hierchization provided by clients which are actually servers provides another level at which the system can be scaled. Scalability is also enhanced by the fact that all the components are for the most part independent from each other and that the communication is message-based and asynchronous, so no part of the system is time-dependent.

6. Quality software design. Well, we will not expose ourselves to potential humiliation by claiming success here. However, we've tried to do as good a job as possible. Our implementation is object-oriented, and we have worked hard at designing well-defined, meaningful interfaces. We've used patterns in several places to great success, and we've taken advantage of some advanced features of C++ (mostly to do with templates) to create generic, type-safe, reusable
components which we feel could be used in other contexts as well. We've taken care to document the interfaces as well as the code itself. And we've written a test suite.

7. **Performance.** Here again, we do not want to prematurely claim success, as we've not used the system long enough or on a large enough scale. Some early experiments which we will describe later, however, produced promising results. Also we have not really worked on optimizing the system, and we're confident that some effort in that direction will certainly improve our numbers.

Now that we've whetted your appetite with this delicious preview of PSolver, you are no doubt eagerly anticipating the main course. Worry not, for you shall not be disappointed. In the next chapter, we will delve into a full discussion of the PSolver architecture and software design.
Chapter 6

PSolver Design

And inside the wheel were spokes of fire

PSolver is a fairly complex system and has a few dimensions that are of interest. To describe it properly, therefore, we will need to take a look at it from different angles and at different levels of magnification. We will discuss the high-level architecture of the system, the PSolver Protocol, the heuristics used for key algorithms, and some of the more interesting code constructs.

6.1 Architecture

6.1.1 Network Topography

Since PSolver is a distributed system, we faced an important decision early on in its design - what should be its network topography. There were two basic choices - a fully distributed system or a master-slave system. The former can have many different actual topographies as far as which nodes interact with which other nodes, but the important thing is that all the nodes are on equal footing - they are fully interchangeable. In a master-slave system, on the other hand, there are different kinds of nodes - worker/slave nodes which perform the actual work on subproblems, and master nodes which coordinate worker/slave nodes. Both models have their own advantages and disadvantages. Fully distributed systems offer better fault tolerance
properties, since there are no special nodes which may crash or become unavailable. They are also more theoretically elegant. On the other hand, master-slave systems are often easier to design, implement and maintain. In addition, such systems are often faster because coordinating decisions are centralized and do not need to be “agreed upon” by all nodes in the system.

We ended up picking the master-slave model, for several reasons. A primary consideration was that it was easier to design and code, since we had limited time and resources. Further, the few existing distributed SAT solving systems used the master-slave model, so we had something to build on rather than having to design from scratch. Also, we expect that interaction between the user and the system will be infrequent, since we expect the system to solve large, time-consuming problems. Therefore the issue of a user being unable to communicate with the system for short periods of time is not important. On the other hand, performance was a top priority, which favored the master-slave model. Finally, we address some of the shortcomings of the master-slave model with state-saving and monitors, which we will discuss shortly.

6.1.2 The Participants

As mentioned earlier, the basic architecture of PSolver consists of four kinds of processes running on a set of machines and communicating with each other. Following the master-slave model, two of the four kinds are server (or master) processes and client (or slave) processes. In addition, there are monitor processes, which help maintain connectivity. Finally, there are user processes. These are not really a part of PSolver as such - they represent the users of the system - but it is convenient to include them when describing how PSolver works.

Figure 6-1 shows schematically a typical PSolver setup. Solid arrows show maintained connections, while dashed arrows show temporary connections. The setup illustrated is one with 4 clients, 2 servers, 3 users and a monitor. Both servers are registered with the monitor. Clients and users use the monitor to obtain server locations, with the result that clients 1, 2, and 3 are connected to server 1 and client 4 and user 3 are connected to server 2.
Figure 6-1: PSolver System Diagram
6.1.3 Interprocess Communication

The processes communicate with each other using a text-based protocol, called the PSolver protocol. In our implementation, the processes communicate over TCP/IP based connections, but the implementation could be easily extended to permit any kind of connection that supports stream-like sending of messages and gives at-most-once packet delivery semantics.

6.1.4 The PSolver Protocol

The PSolver protocol is a simple set of conventions that defines the style in which information to be sent from one process to another is to be encoded, enumerates the types of messages that can be sent between processes, and specifies the expected responses to each message. Each message in the PSolver protocol basically consists of a message type, and a number of parameters that carry the actual information. The PSolver system is completely defined by this protocol, which makes it very easy to extend the system by creating new implementations.

6.1.5 Load Balancing

The other major problem we had to solve is how to achieve load balancing. In the context of distributed SAT solving, this translated into figuring out how to break up the original SAT instance into subproblems to give to different clients, and how to ensure that each client had work to do. This had to be achieved given that clients could disconnect in the middle of solving a subproblem, and that new clients could connect while a problem was being solved.

Our solution is based partially on the method used by PSATO, but with a new twist. Like PSATO, we use guiding paths to describe subproblems and to assign those subproblems to clients. Unlike PSATO, however, we do not prefabricate a list of guiding paths and create new ones at the server by splitting some of the ones in the list. Rather, whenever a client finishes solving a subproblem, or a new client connects to the system, the server picks a client currently working on a subproblem...
and asks it go give up a piece of the subproblem it is currently using. The client can do this in any way it chooses. The client then sends to the server a guiding path representing the subproblem it gave up and a guiding path representing the subproblem it is still solving. The two guiding paths must add up to the subproblem the client was originally solving. The server then assigns the guiding path that the client just gave up to the idle or new client.

This approach offers several advantages over the more static algorithm employed by PSATO. First, it deals more smoothly with varying number of clients. PSATO assumes that the number of clients is known at the start and does not change over the solution of a SAT instance. This is not the case in PSolver - clients can connect to a server while it is working on a SAT instance. This makes it inconvenient to maintain a prefabricated list of guiding paths of a certain length. Second, it is more dynamic, since instead of deciding beforehand how the problem should be split up into guiding paths, it does the splitting only when it is required and it splits the current guiding path of a client. This prevents the server from splitting up the problem into too many pieces if that is unnecessary. Finally, it allows flexibility in determining which subproblem will be split up (decided at the server) and how that subproblem will be split up (decided at the client whose subproblem is split up). This allows use of heuristics based on current information to make better decision, and also allows different clients to use different heuristics to make splitting decisions. This is useful since different clients might use different algorithms for solving subproblems and thus are better able to decide how best to split their subproblems.

As mentioned earlier, we use guiding paths to represent subproblems in server-client communication. We described guiding paths in the chapter on PSATO. Since guiding paths play a central part in the system, and figure prominently in more detailed description of the system later on, it is important to have a good understanding of them. So, a review of the section on guiding paths (chapter 4, section 3) is encouraged before continuing.
6.2 High-level algorithms

We will now describe in more detail each of the four processes, and the basic algorithms they run.

6.2.1 server

The server is the hub of the PSolver system. It accepts SAT instances to solve from users, breaks them up into subproblems, assigns them to clients to solve, keeps track of the clients' progress, aggregates the results and reports them to the user. The server can handle clients connecting and disconnecting at any time, and it allows multiple users to connect and can thus be solving multiple instances at any given time, although any given user can only have one instance being solved by the server at a time. A user can cancel the solution of its instance at any time. Also, the server periodically saves the state of the solution for each instance it is working on, so that if the server crashes or a user becomes disconnected from the server, the user can subsequently request that the server resume computation.

The basic algorithm run by the server is a message processing loop as follows:

```
register self with monitor
DO
get next message
process message
UNTIL message is user_quit
unregister self with monitor
```

The registering/unregistering with the monitor will be explained further in the subsection describing the monitor. The message-processing step is accomplished by a state-machine algorithm which is given later on when implementation details are described. However, it is important to describe a few algorithms at a high level, most importantly the load balancing algorithm.

Basically, the server maintains 3 lists: a map of active clients to the subproblems they are working on, a list of pending subproblems (subproblems that still need to be
solved but are not currently being worked on), and a list of idle clients - clients that are not currently working on any subproblem. Then the algorithm is:

**WHEN a client becomes newly idle:**
- if the pending-subproblem list is not empty, remove one of the subproblems from it and assign to the client. Add this client-subproblem mapping to the client-subproblem map.
- otherwise, if the active client-subproblem map is not empty, pick a client-subproblem pair and ask the client to give up a subproblem. Meanwhile, place the newly idle client on the idle client list.
- otherwise, there is nothing currently to do. Add the client to the idle-client list.

**WHEN a new subproblem becomes available:**
- if the idle-client list is not empty, pick a client off it, assign the subproblem to it, and record the association in the active client-subproblem map.
- otherwise, add the subproblem to the pending-subproblem list.

In this algorithm a newly idle client is either a client that has just connected to the server or a client that had just finished solving a subproblem. A new subproblem can become available in one of several ways: either the server has just been assigned a new problem by the user, a client has just given up a subproblem, or a client had died and the subproblem it had been solving now needs to be assigned to another client. The above algorithm handles these distinct cases in a uniform manner.

The above algorithm description leaves open two questions: first, when the pending-subproblem list is not empty and a client needs a subproblem, which pending subproblem is chosen; second, when the pending-subproblem list is empty and the server needs to issue a give-up request to one of the clients, how does it choose which client to issue the request to. Our solution is that we always try to assign the longest subproblem possible. The longest subproblem is one with the earliest open pair in
its guiding path, with ties resolved by the most open pairs in the guiding path, and further ties resolved at random. This heuristic is used to both pick the subproblem from the pending-subproblem list, and to pick the client to issue the giveup request to - we pick the client which is working on the longest sub-problem. The goal of this strategy is to minimize the number of times clients become idle and have to wait for a subproblem.

The only complication to this algorithm arises because a client can refuse a give-up request (as explained in the section on the client). We deal with this by simply picking a new client according to the same rule and issuing a new request.

Figure 6-2 illustrates a portion of the sequence of interactions between the user, server and clients for solving a typical problem. Here, the user and clients are already connected to the server. The user gives the server a problem to solve. The server gives one of its clients the entire problem to solve, then asks it to do a giveup. The client responds to the giveup, and the server assigns the subproblem given up to the second client. After both clients return UNSAT, the server returns to the user the result UNSAT. The simplifying assumption here is that both clients return UNSAT simultaneously. In a complete sequence of interactions, there would be several rounds of giveups, as one of the clients completes its subproblem while the other is still working on its.

Another algorithm that deserves mention here is the state saving algorithm. As mentioned earlier, state-saving allows us to save the state of the computation for a SAT instance. If the server crashes at some point, it can, once it restarts, resume solving the instance from where it left off, instead of having to start from scratch. This property is very important, particularly when we expect our system to handle very large instances requiring days of computation to solve.

It turns out that using guiding paths to describe subproblems, it is very easy to save the state of the computation - one simply needs to save all of the guiding paths that still need to be examined. In the context of our server algorithm, this simply means all the guiding paths on the pending-subproblem list, as well as the guiding paths in the active client-subproblem map. By periodically saving these guiding
Figure 6-2: PSolver: Interaction between server, user, and clients during solving of problem
paths, state-saving is accomplished.

Finally, we should discuss solution enumeration. Solution enumeration allows the system to, for a SAT instance which is satisfiable, to list all the solutions. This is a very useful feature, in particular for applications such as model checking, where different solutions can represent different models that match the given criteria. With the guiding paths used to represent subproblems, solution enumeration becomes easy - it almost comes for free. When a client finds a solution, that solution can be represented as a guiding path - in fact a guiding path that includes all the variables in the instance. This guiding path still retains information about the state of the search when the solution was found. Thus this guiding path also represents a subproblem, and so can be used to continue the search where it left off when the solution was found.

To accomplish solution enumeration, then, the following algorithm is used.

IF a client finds a solution (represented by a guiding path)
- send server_cancel messages to all other clients working on subproblems
- collect response client_result messages from these clients.
  These messages contain guiding paths representing subproblems these clients had left to solve when they received the server_cancel message
- add all of these guiding paths to the pending-subproblem list.
  Also add the guiding path representing the solution, first ‘‘incrementing’’ it. When the user asks for the next solution, start solving as normal

The part about “incrementing” the solution guiding path deserves further explanation. Since that guiding path is a solution, we cannot use it directly - if we assign it to a client it will simply return the solution again. Thus we need to modify it to create the guiding path which represents the next step in the search.
6.2.2 client

The clients are the workhorses of the system. This is where the actual work takes place. Essentially the client is just a thin wrapper around a sequential SAT solving algorithm. This wrapper makes it possible for the client to locate and connect to servers, receive messages from the server and send appropriate responses to it, after interfering with the underlying SAT solving algorithm appropriately. For example this wrapper must ensure that give-up requests are handled properly.

The high-level algorithm run by the client is:

DO
  connect to a monitor
  DO
    request a server location
  UNTIL monitor sends a working server location
  connect to server
  send client_new message to server
  receive server_ack message from server
  DO
    get message from server
    process message
  UNTIL server dies or sends quit message
  disconnect from server
UNTIL client process is killed

Here, again, the interaction with the monitor is explained in the section about the monitor. The message processing algorithm is given below:

IF message is server_solve
  - start solving the subproblem given in the message
  - when done solving the problem, send client_result message to server with the result
ELSE IF message is server_cancel
  - stop solving the subproblem currently being solved.
  - send back to server the guiding path representing subproblem remaining to be solved
ELSE IF message is server_giveup
  IF we want to give-up part of the subproblem being solved
    - split subproblem in two. continue solving one of the two new subproblem. send message to server telling it the subproblem given up and the subproblem the client is now working on
  ELSE
    - send client_giveup_refuse message back to server telling it we refuse to giveup.

The interesting question, of course, is how does the client decide whether or not to fulfill the request to give-up a subproblem. The idea here is that this decision should be made with the help of the underlying SAT solving algorithm and be based on how close the algorithm is to finishing the solving the subproblem it is working on. This question can be very difficult to answer, however, and so we use a simple heuristic instead: we simply refuse any give-up requests until the client has been solving the subproblem for some constant prespecified amount of time (typically a few seconds). This ensures that if a subproblem is really small, the client will solve it completely without splitting it up. This is good enough to solve a problem we encountered with early versions of our system (which we call 'thrashing'): when it would get close to solving a problem, the subproblems became smaller and smaller, a self-perpetuating process since as the subproblems would get smaller, clients would finish solving them quicker and so request new subproblems which would require other clients splitting up the subproblems they were working on. Eventually the overhead time required to split up a subproblem, assign it to a client, and the initialization time for SAT solving algorithm would overwhelm any savings from parallelization and the system would significantly slow down towards the end. The solution of working for at least a few
seconds on any subproblem before agreeing to give up a piece of it ensured that small subproblems did not get split up further and eliminated this undesirable behavior.

6.2.3 monitor

The monitor serves as the connectivity link between servers and clients/users and as a tool to aid fault tolerance. We expect that servers, clients and users will all crash, as they run complex algorithms. In addition one can expect that the number and location of servers, clients and users will change. What is needed, then, is a way for servers and clients/users to locate each other and maintain connectivity. This is accomplished through the monitor.

The monitor is a very simple process that is expected to always be running at a specified location. Servers register with the monitor when they start running and unregister when they stop running. Whenever a newly started client or a user needs to find a server to connect to, it queries the monitor. If at any time the server crashes, the client can request a new server to connect to.

This allows us to build what we call virtual SAT solving clusters. Any person who wishes his machine to become part of the cluster can download and install the client on his machine. He then configures the client with the location of a monitor (or several monitors). That’s it - no further interaction is required on the part of the person. The client will obtain server locations from the monitor, connect to them, and make itself available for solving subproblems. If the client becomes disconnected from the server, it will simply get a new server to connect to from the monitor. If there are none, it will wait until one becomes available.

The monitor also runs a message-processing loop algorithm. It is very simple and given below.

\[
\begin{align*}
&\text{DO} \\
&\text{accept message} \\
&\text{process message} \\
&\quad \text{IF message is server_register}
\end{align*}
\]
add server to list of active servers
ELSE IF message is server_dead
    remove server from the list of active serves
ELSE IF message is client_server_request
    IF list of active servers not-empty
        pick a server from the list and send a monitor_server message
        back with the server information
    ELSE
        send a monitor_server_none message back
ELSE IF message is monitor_server_list_request
    send a monitor_server_list message back with the list of active
    servers
WHILE true

Notice that the monitor can also perform a load-balancing function. This is be-
cause it can pick any server from its list in response to a query for a server. This can
be used to ensure that each server has roughly the same number of clients working
for it and roughly the same number of users.

Figure 6-3 illustrates the use of monitors by clients and users to discover server
locations. First the server registers itself with the monitor. Later, when a client and
a user ask for a server, the monitor responds with the location of the server.

6.2.4 user

The user is not really a part of the system as such - it is a client of it. However it
is convenient to include it when discussing the interactions between different com-
ponents of the system. Also, since to make use of the system one must be able to
communicate with it using the PSolver protocol. Thus we can think of the user pro-
cess as that layer which sends and receives PSolver Protocol messages from the rest
of the system. This layer may be acting on behalf of another program, or it may be
acting on behalf of a human user.
Since the user is not a proper part of the system, there is no algorithm to describe. The messages the user needs to send to make use of the PSolver system, and the response messages it receives with the results are described in the section on the PSolver protocol.

6.3 PSolver protocol

As mentioned earlier, the system is completely specified by the PSolver Protocol. The PSolver Protocol comprises a set of messages and the semantics governing those messages, that is the context in which those messages can be sent and the expected responses to the messages. We now describe the protocol. We focus here on semantics. For each the message we give its name, the arguments it carries, the meaning of the message and the context in which it is sent, the expected response messages to the message, and additional notes. The syntax of the protocol is given in an appendix.

The PSolver protocol consists actually of three subprotocols: the client-server
protocol, the user-server protocol, and the monitor protocol. We describe each in turn, in a table at the end of this chapter.
6.4 Software design

Our implementation of PSolver was done in C++. There are several important sub-systems that make up the codebase for PSolver which we will now describe.

6.4.1 The SendableObject Hierarchy - lightweight serialization

Since we chose not to use third-party software such as CORBA to implement inter-process communication, we were forced to so ourselves. This raised a few important problems. One of these was how to send complex datatypes over the network - essentially we needed a lightweight method for serializing objects and recreating them on the other end. Our solution was to have an object hierarchy and use the prototype pattern. At its root is the SendableObject class, Its (abridged) declaration is:

```cpp
class SendableObject {

public:

    SendableObject (istream& data);
    SendableObject (const SendableObject& other);

    virtual void toString (ostream& out) const = 0;
    virtual SendableObject* clone () const = 0;
    virtual SendableObject* sameKind (istream& in) const = 0;
};
```

The important methods here are the toString method which writes a textual representation of the object to the output stream, and the sameKind method which returns an object represented by the textual representation present in the input stream argument. Both of these are pure virtual methods and are therefore defined
in subclasses of SendableObject to do the right thing. In particular, in the case of sameKind, this means returning an object of the right subclass of SendableObject.

Any object which must be sent over the network must be a subclass of SendableObject. In addition to defining the above pure virtual methods, there are a few extra requirements for such classes. A class Foo which is a subclass of =verb—SendableObject—must satisfy the following:

- it must have a non-public no-arg constructor
- it must have a constructor which takes a const istream& argument
- it must have a static public member of type Foo.

The static member is necessary for the MessageRegistry described in the next section. The textual representation of the object must be done according to the rules described in the appendix on the PSolver Protocol syntax.

We do not deal with complex issues such as templatized data types. Thus if we have, for example a type Literal which is a subclass of SendableObject and we wish to send a list<Literal> over the network, we'll need to create a new type LiteralList which is again a subclass of SendableObject. However, this does give us a way of getting lightweight serialization for a very low cost.

### 6.4.2 Message, MessageSet and MessageRegistry - parametrized messaging

To represent PSolver Protocol messages, we needed a type that would support having a variable number of parameters with varying types. The Message class is a subclass of SendableObject which is abstractly defined as a message name and a list of named arguments. The important parts of the class declaration are:

```cpp
class Message : public SendableObject {
    friend MessageRegistry;
};
```
public:

typedef map<string, SendableObject*> argMap;

Message () { }
Message (const string& _name, int _clientId, const string& _problemId,
    SendableObject* arg = NULL);
Message (const string& _name, int _clientId, const string& _problemId,
    argMap &args);
Message (const Message &other);
Message (istream& data);
virtual ~Message ();

const string& getName () const { return name; }
SendableObject* getArgument () const; // get default arg
SendableObject* getArgument (const string& name) const; // get named arg

SendableObject*& operator[] (const char* argName)
    { return args[argName]; }
SendableObject* operator[] (const char* argName) const
    { return args.find (argName)->second; }

protected:
    bool correctFormat (Message* prototype) const;
    void initFromStream (istream& in);

private:
    typedef argMap::iterator args_iterator;
    typedef argMap::const_iterator const_args_iterator;
Note that clientId and problemId are special-case arguments and are dealt with separately. This is because those arguments are used internally by the communication package to route messages to appropriate destinations. The getArgument methods have a return type of SendableObject*, which can then be cast by the client to the appropriate type. As a result, all message arguments must be of a type which is a subclass of SendableObject. For built-in types we provide wrapper classes that transparently (using the C++ typecast user-defined operator construct) adapt them to the SendableObject hierarchy.

The important methods here are initFromStream and correctFormat. The former initializes a Message object based on a textual representation present on the given input stream. This is rather tricky, because a Message object is composed of objects of different types, depending on the type of the message that the Message object represents. To properly initialize itself it needs to figure out what these types are. There are a few different ways to do this. One is to include type information in the textual representation of a Message object, or in the textual representation of SendableObject's in general. However, since C++ has no built-in global run-time type information system, like Java's, this would be difficult to do in a consistent manner. Instead, our solution was to use the Prototype pattern. Each subclass of SendableObject has a static member prototype which is a prototype object of that class. Then, for each possible PSolver Protocol message, we create a prototype Message object. These prototype Message objects are stored in a class called MessageRegistry. Then, when a message arrives, we first look at the type of the message. We then retrieve the appropriate prototype Message object from MessageRegistry. This tells us the class types of the arguments of this message, and we can use those to construct the actual Message object corresponding to this message.
To facilitate the management of the prototype Message objects, we have a special
file called messages which contains the definition of all the messages in the PSolver-
Protocol - for each message, its name and the names and types of its arguments are
specified. Messages are broken up into message sets, corresponding to subprotocols.
We then have a utility that reads this file and generates the C++ code to create and
populate the MessageRegistry.

The messages file also contains definitions of all the types used in the PSolver
Protocol - these types are described further in Appendix B. These are the types that
correspond to subclasses of SendableObject. In the future we plan to write a utility
that will generate the definitions of these subclasses directly from the messages file,
but currently this is done by hand.

6.4.3 CommunicationObject - generic interface to message
passing

We wanted to abstract from the particular transport protocols and hardware used
to send messages between the different components of the system. This is where the
CommunicationObject hierarchy comes in - it is an interface to sending and receiving
Message objects. At the root of the hierarchy is the class CommunicationObject,
which simply defines methods to send and receive Message's:

class CommunicationObject {
    public:
        CommunicationObject () { }
        ~CommunicationObject () { }

        virtual void sendMessage (Message *what) = 0; // send message to peer
        virtual Message *getMessage () = 0; // get message from peer

        virtual bool hasMessages () = 0; // do i have pending data?
};

71
There are further subclasses which add to the interface for more specific cases. For example, ServerCommunicationObject and ClientCommunicationObject add methods specific to server and client behavior, respectively. BufferedCommunicationObject adds buffering concepts, while ThreadedCommunicationObject adds the idea of running the communications system in a separate thread. Finally, there are specific classes, which implement the interface for sockets communication. However, it would be easy to add implementations which change the transport method, add encryption, and the like.

We actually plan to change the CommunicationObject hierarchy to make it conform to the iostreams C++ standard library. This will make it easier to extend and to mix and match implementations.

6.4.4 Event/EventChecker/Callback/EventManager system
- generic parameterized event handling

The server component in our system is difficult to conceptualize and program, because it must deal with things in a very asynchronous manner. This is due to our choice of message-passing, as opposed to, say, remote call invocation, as the distributed communication paradigm of choice. Because of that, we cannot treat commands to remote processes as function calls which return when completed. Thus, if we want to for example tell a client to stop solving its subproblem, we cannot simply do something like

```cpp
// blah blah blah
client.cancel(); // tell client to cancel
freeClientPool.addClient(client); // return client to free client pool
// do other stuff
```

Instead, we send a message to the client telling it to stop working. At some point later, we will hopefully get a message back saying that the cancellation did happen and the the guiding path at the time was such and such. But in the meantime, the
status of the client in the eyes of the server is in a sort of intermediate state - neither working nor free.

Another issue is that the server can be working on several problems at once, coming from several users. It must somehow coordinate between these different users, since it has a single pool of clients available to work on subproblems coming from any of these problems.

The result is that the logic of such a server implementation becomes pretty hairy. To cope with this, we programmed the server as an event-driven system. There are a variety of objects which are involved in the server - each problem has a corresponding ProblemManager object, there is a Coordinator object which manages server-wide resources such as the client pool, there is a ClientReaper object which deals with clients that we working on problems which have become discarded (because, for example, the user who posed the problem disconnected from the server), there is the MessageReceiver object which is the interface to the CommunicationObject subsystem. These objects must all interact with each other. Instead of directly calling each others methods, which would create a nightmare as far as having to keep references and understand interaction diagrams, they do so through a central clearinghouse - the EventManager.

When an object wants to do something that requires interaction with the other objects - for example a ProblemManager wants to release a client so it can be used by other ProblemManagers - it creates an Event object, puts the appropriate information in it - in this case the client ID - and tells EventManager object to generate the event. Event consists of an event name and a variable number of named parameters of arbitrary type. It is thus similar to Message except it differs in that the types of parameters are completely arbitrary, but the number of arguments is actually limited to between 0 and 3. Objects which are interested in particular events register with the EventManager by providing a Callback and a EventChecker object. The EventChecker object can be used to put in place further rules about which events the object is interested in - it can examine the Event parameters and implement a restriction on those parameters’ values. The Callback object has a method which
will be invoked if an event of the appropriate type is generated. It is basically an adapter - we provide implementations of Callback which end up calling a global or class method and translate Event parameters into method arguments.

This subsystem demonstrates and makes effective use of the C++ template system and the C++ STL library. The former allows seamless creation of completely generic Event, Callback and EventChecker objects and allows them to work together in a type-safe manner without the need to resort to any casting. The latter allows us to easily compose different objects so that they work together in a natural way. In all, this was a very successful piece of software design and can be easily adapted for use in other applications.

Next, we abstract from the PSolver system in particular, and SAT solving in general, and discuss the lessons we learned about building distributed, platform-independent systems in general.
Message Name: user_solve
Arguments: The problem to be solved
Meaning & Context: gives the server an instance to solve. This is usually the first message sent by user to server after the user connects to the server.
Expected Response: server_user_ack message from server to user to acknowledge receiving the problem, followed later by a server_result message with the solution
Notes: This can only be sent if the server is not already solving a problem for this user. This means it can be sent right after the user connects, or after the server sends the user a server_result message with the UNSAT result, or after the user sends the server a user_done message for the previous problem being solved.

Message Name: user_resume
Arguments: problem ID for the problem to be resumed
Meaning & Context: A request to resume solving a problem which was not completely solved. The context for this is the same as for the user_solve message.
Expected Response: server_user_ack message
Notes: Notes for the user_solve message apply here. Also in the original user_solve request the user must have specified that he wanted state saving to be active

Message Name: user_cancel
Arguments: user’s ID
Meaning & Context: requests that the server stop working on the problem the user previously assigned it
Expected Response: a server_result message with the CANCELLED result.
Notes: The user must previously have assigned a problem to the server and the server has not completed solving that problem.

Message Name: user_next
Arguments: user’s ID
Meaning & Context: requests that the server find another solution for the problem the user has assigned the server. This allows solution enumeration, i.e. if the instance is satisfiable and has multiple satisfying assignments, this allows the user to get all of these assignments from the server.
Expected Response: nothing immediately, eventually a server_result message from the server with the next solution or the UNSAT result.
Notes: This is only valid if the previous server_result message was a SAT or a CANCELLED result

Message Name: user_done
Arguments: user’s ID
Meaning & Context: indicates that the user is finished with the problem it has assigned to the server, so the server can stop keeping track of it. This message is necessary only if the previous server_result message was a SAT or a CANCELLED result.
Expected Response: none
Notes:
Message Name: *user_quit*
Arguments: none
Meaning & Context: tells the server to stop working on all the problems it is currently working (including those of other users), and to stop running. This is a way to remotely shut down the server. It is essentially equivalent to receiving a user_cancel message from all of the users currently connected to the server.
Expected Response: a user_result message with the QUIT result if the server is currently working on a problem for this user, nothing otherwise.
Notes: this causes the server to send user_result messages with the QUIT result to all the users connected to it.

User-server subprotocol: server to user

Message Name: *server_user_ack*
Arguments: the problem ID and the user ID
Meaning & Context: this is sent in response to a user_solve or a user_resume message. It assigns an ID to the user which is then used to identify that user, and, if in response to a user_solve, assigns an ID to the problem and gives it to the user so that it can later use it to refer to the problem in a user_resume message. If it is in response to a user_resume message, the problem ID simply the same as given by the user in the user_resume message.
Expected Response: None
Notes:

Message Name: *server_result*
Arguments: the user ID, the result type, and depending on the result type, further information about the result. Possible result types are UNSAT, meaning the instance is unsatisfiable, SAT meaning the instance is satisfiable, in which case a solution is included, CANCELLED, meaning the user told the server to cancel solving the problem, QUIT, meaning the user told the server to quit, and ERROR, meaning an error occurred, in which case a string is included describing the error.
Meaning & Context: reports the result of solving the instance to the user. If the result is UNSAT or ERROR, then work on the problem is considered finished and the user can assign another instance to the server. If the result is SAT or CANCELLED, then the problem is still active and the user can choose to request more solutions or indicate that it is done with the problem. Finally, if the result is QUIT then the server is about to stop running.
Expected Response: None if result is UNSAT, ERROR, or QUIT. user_next or user_done if result is SAT or CANCELLED.
Notes:
Server-client subprotocol: client to server

Message Name: **client_new**
**Arguments:** client information - describes client capabilities
**Meaning & Context:** sent by client right after it connects to the server - lets the server know the client is ready to solve subproblems and allows the client to describe itself to the server.
**Expected Response:** either server_problem_solve which gives the client a subproblem to solve, or, server_ack which acknowledges the client and says that the server does not have a subproblem for it to solve yet.
**Notes:** The client information argument is mostly for future extensibility and allows the client to describe special capabilities it supports so that the server can treat it appropriately.

Message Name: **client_result**
**Arguments:** the client’s ID, the problem ID, and the result. the result can be UNSAT, meaning that there is no solution in this subproblem, SAT meaning it found a solution in which case the solution is included, CANCELLED meaning the client received a server_cancel command while solving the subproblem, in which case the client a guiding path is included representing the subproblem the client had left to solve at the time of getting the cancel message, QUIT meaning the client received a server_quit command while solving the subproblem, TIMEOUT meaning that the specified timeout period expired without the client finding a solution, or ERROR meaning some error occurred in which case a description of the error is included.
**Meaning & Context:** used by client to report the result of solving a subproblem, or to acknowledge a cancellation request. Indicates also that the client is now idle and ready to receive another subproblem to solve.
**Expected Response:** None immediately
**Notes:**

Message Name: **client_progress**
**Arguments:** client’s ID, problem ID, and a guiding path.
**Meaning & Context:** used by client to notify the server of its progress on the current subproblem. The guiding path sent represents the part of the subproblem still left to solve. This allows the server to more accurately keep track of the clients’ progress.
**Expected Response:** None
**Notes:** Progress messages are sent at regular intervals specified by the server when it first assigns the client a subproblem.

Message Name: **client_status**
**Arguments:** client Id, problem ID, a status string
**Meaning & Context:** used by the client to send a generic status report to the server. this allows the client to send various debug messages and status messages and the like.
**Expected Response:** None
**Notes:**
Message Name: client_giveup
Arguments: client’s ID, problem ID, a guiding path representing the subproblem given up, a guiding path representing the new subproblem the client is now solving.
Meaning & Context: this is sent by the client to the server to give up a part of the subproblem the client was solving at the time, in response to the server’s request to do so (the server_giveup message). The client sends the subproblem it has given up (which the server can assign to another client) and the subproblem it is now solving.
Expected Response: None
Notes: The two subproblems add up to the subproblem the client had left to solve until receiving the giveup request

Message Name: client_giveup_refuse
Arguments: client’s ID, problem ID
Meaning & Context: used by the client to refuse a giveup request. The client does this when it thinks it’s very close to solving its subproblem and so giving up a piece of it does not make sense.
Expected Response: None
Notes:

Message Name: client_file_request
Arguments: client’s ID, problem ID
Meaning & Context: used by the client to request the file containing the instance to be solved if it cannot access it
Expected Response: a server_file_send message with the instance file
Notes: if the client and server share a file namespace and the permissions are appropriate, then the client can access the instance file directly, without having to request it from the server. If not, this command is available for the client to request the file to be sent over its connection with the server.

Client-Server subprotocol: server to client
Message Name: server_ack
Arguments: client ID
Meaning & Context: Used by the server to acknowledge a new client and assign it an ID if the server does not have a subproblem for it to solve yet.
Expected Response: None
Notes:

Message Name: server_problem_solve
Arguments: client ID, problem ID, the subproblem
Meaning & Context: used by the server to assign a subproblem to the client. The subproblem consists of a guiding path describing the subproblem and parameters specifying the timeout period (how long to work on the subproblem before giving up) and the progress interval (how often progress messages are expected)
Expected Response: None immediately, eventually a client_result
Notes: the client must not already be working on a subproblem
**Message Name:** server-problem-giveup  
**Arguments:** client ID, problem ID  
**Meaning & Context:** used by the server to request that the client give up a piece of the subproblem it is solving  
**Expected Response:** either a client.giveup message with the subproblem given up or a client.giveup.refuse message indicating the client refuses to give up a subproblem  
**Notes:** The client must be currently working on a subproblem

**Message Name:** server-problem-cancel  
**Arguments:** client ID, problem ID  
**Meaning & Context:** used by the server to tell the client to stop working on the subproblem it is currently solving  
**Expected Response:** client_result with the CANCELLED result and the guiding path representing the subproblem the client had left to solve when it received the cancel message.  
**Notes:** the client must currently be working on a subproblem

**Message Name:** server-quit  
**Arguments:** clientId, problem ID  
**Meaning & Context:** used by the server to tell the client it is about to stop running and therefore to stop working on its current subproblem. This is essentially equivalent to the server-problem-cancel command except that after the responding the client is free to disconnect from the server.  
**Expected Response:** a client_result message with the QUIT result  
**Notes:**

**Message Name:** server-file-send  
**Arguments:** client ID, problem ID, the SAT instance  
**Meaning & Context:** used by the server to send the client the SAT instance for which the server wants the client to solve a subproblem, in response to a client_file_request message  
**Expected Response:** None  
**Notes:** the client must have sent the server a client_file_request message
Monitor subprotocol: server to monitor

Message Name: server.register

Arguments: server info - describes the location of host (i.e. hostname, port number)

Meaning & Context: sent by server to the monitor when the server starts running; registers the server with that monitor so that subsequent queries to the monitor for servers may return that server

Expected Response: None

Notes:

Message Name: server.dead

Arguments: server info - same as for server.register

Meaning & Context: sent by server when it is about to quit or crash to the monitor to tell the monitor the server is no longer running and so should not be returned in response to queries

Expected Response: None

Notes: this can also be sent by a client or a user to the monitor if that client or users discovers that a server is dead

Monitor subprotocol: client or user to monitor

Message Name: client.server.request

Arguments: None

Meaning & Context: sent by a client or a user to request a server location to connect to

Expected Response: a monitor.server message with a server location or a textttmonitor.server.none message if the monitor doesn’t know about any active servers

Notes:

Message Name: monitor.server.list.request

Arguments: None

Meaning & Context: sent by a client or a user to request the list of all active server locations the monitor is aware of

Expected Response: a monitor.server.list message with the server list

Notes:
Monitor subprotocol: monitor to client or user

**Message Name:** monitor_server
**Arguments:** client ID, server info
**Meaning & Context:** sent by monitor in response to a request for a server location. the server info argument gives the location of an active server.
**Expected Response:** None
**Notes:**

**Message Name:** monitor_server_none
**Arguments:** client ID
**Meaning & Context:** sent by monitor in response to a request for a server location, if the monitor doesn't know about any active servers
**Expected Response:** None
**Notes:**

**Message Name:** monitor_server_list
**Arguments:** client ID, list of server info's
**Meaning & Context:** sent by monitor in response to a request for the list of all active servers it knows about. sends the list
**Expected Response:** None
**Notes:**
Chapter 7

A step back: Why was it hard

Singing the blues

In this chapter we ignore for the time being the main topic of this paper - SAT solving - and focus instead on the details of writing a distributed, platform-independent system of any kind. Our do-it-yourself approach to implementing PSolver allowed us to gain some valuable insight into just why it is difficult to build such systems, and we wish to share this insight. We discuss separately the challenges of building a distributed system, and the challenges of building a platform independent one.

7.1 Why distributed is hard

There are many reasons why building distributed systems is difficult. Inherently, such systems are more complex than sequential ones, since not only must one think about the code flow of one process, but the interactions of multiple processes. Here we focus on four particular challenges that we faced in our system.

7.1.1 Difficult to apply/visualize design patterns

A recent trend in object-oriented software development is the extensive use of design patterns. Design patterns are essentially pre-fabricated solutions for particular design problems, adaptable to each particular case. They allow the designer to not have to
reinvent the wheel when faced with common design problems - in a way they are the equivalent of algorithms and data structures for problem analysis and design. Using design patterns can significantly shorten design times and eliminated incorrect designs. Indeed in our own system, we use design patterns extensively in solving certain design problems. Some of these patterns were discussed in the last chapter.

However, most design patterns assume a sequential computation model and are not applicable to distributed computation problems. While there are design patterns that cater to distributed computation, these patterns are often more difficult to adapt to particular situations, and are harder to visualize, since time, which is not important for sequential design patterns, becomes a factor.

7.1.2 Type systems are difficult to implement

One of the most difficult things about building a distributed system is correctly maintaining type information across all processes in a distributed system when objects need to be sent from one process to another. The problem is this: given that an object needs to be converted to a textual/binary representation to be sent over the network, and then the object needs to be recreated on the other end, enough type information must be included with the object representation to allow the other side to re-create the object of the correct type. Certain languages, such as Java, have built-in support for this and therefore eliminate the issue. However, in C++, only very limited run-time type information is available. When template classes are introduced, the problem becomes even more difficult since a solution such as a per-class type identifier field becomes insufficient.

A further difficulty arises when one considers the possibility of various components of the distributed system being developed separately, and therefore the possibility that there are different “versions” of the various components and sometimes the version numbers of the components in a system might not match. This includes the possibility that class definitions change from version to version.
7.1.3 Memory management is difficult

Related to the type information problem is the difficulty of correctly managing memory in a distributed system. Once again, in a language like Java, which has built-in garbage collection, this is not a problem. In C++, however, the issue is more problematic. The basic problem is: how to manage memory for the objects that are sent over the network. When these objects are re-instantiated on the receiving end, when should they be deallocated, and whose responsibility should it be for doing so. An inappropriate memory management scheme can result in either excessive memory leakage or excessive copying of memory which can slow down a program significantly.

7.1.4 Testing is difficult

Finally, a very important issue in distributed systems is testing. The problem is that the most common and the most difficult-to-track down bugs in distributed systems arise from timing issues - the order in which messages are sent and received, for example. Unfortunately, it is very difficult to control this - it is virtually impossible to replicate interactions within a distributed system so that their timing is the same from one run to another. This makes it very difficult to accurately test such systems.

7.2 Platform independence

While the difficulties in building distributed systems are largely conceptual, those in writing systems that are platform independent are more practical and mundane, or at least they were so in our case. However that does not make them any less frustrating or less of a drain on design and implementation efficiency. We focus on three specific issues:

7.2.1 Differences in implementation of same functionality

Often, the same functionality is available on different platforms. However, there are small difference in how that functionality is implemented, and in the interface
offered for using that functionality, that need to be accounted for. For example, both Windows and UNIX offer socket interfaces. The windows version (Winsock) is very similar to the UNIX version (Berkeley sockets) and the API’s for the two are virtually identical. However, there are several minor differences in the semantics of some function calls and in the error reporting offered by the two versions. These differences, if unnoticed, can make the code behave incorrectly or crash. To account for the differences, one must write wrapper code that does the appropriate thing based on system type.

7.2.2 Unavailable functionality

On the other hand, sometimes functionality available on one platform will be completely missing from another platform. An example of this is signals - while Unix has them, and many Unix programs make use of them, Windows has nothing that is close. This can be a nasty problem, especially if you already have code which relies on such functionality and are trying to port this code to other systems. The only solutions are to either avoid writing code that relies on anything remotely low-level, or write two different implementations of the same higher-level functionality, each of which relies on different low-level mechanisms offered by different platforms. The former sometimes is virtually impossible, as some kinds of things can only be done by using low-level API’s. The latter leads to software that is difficult to maintain, as any changes to the higher-level functionality must be done for each of the system-dependent implementations.

7.2.3 Buggy portability tools

Of course one can avoid these problems altogether by using a portability package, which provides the API’s offered by one platform on another platform. One such portability tool is Cygwin, which claims to offer a full UNIX API for the windows platform, and which we use in PSolver. Unfortunately, such tools often are either commercial (and must be therefore paid for, and come with licensing issues), or come
from the free software crowd, and have bugs. The latter is the case with Cygwin. While for the most part it does an excellent job and proved very useful. However, it turned out to have a bug in its implementation of UNIX sockets that caused our server component not to accept any incoming connections. Because the people who work on the free software do so voluntarily, bug fixes are slow to arrive.

This leads us to our final cautionary tale about using compatibility tool such as Cygwin. This is basically a rant to let out the frustration of dealing with some very annoying problems so feel free to skip it and go on to the next chapter. As mentioned before, the UNIX sockets implementation in Cygwin is buggy. We therefore were forced to port our sockets-based code to Winsock to have it run on Winsock. As it turns out, Winsock is very similar to UNIX sockets, but with some minor and very annoying differences. More annoyingly, the implementation of Winsock uses certain global variables and methods (betraying its origins) that are essentially generic UNIX variables and methods. However, since they are redefined in the Winsock implementation, this makes it very difficult to compile it with the Cygwin code which includes these same variables and methods as part of its general UNIX code base. The result was that in order to get the whole thing to compile, very careful conditional computation guards were necessary in order to ensure only one set of declarations and definitions was used. This was very annoying, resulted in unnecessarily jumbled code, and took a disturbingly long time to get right. So the lesson is: even seemingly simple things can get difficult when it comes to system independence.
Chapter 8

Results

Was it all worth it

All this effort would not be worth much if we didn't have some actual data to present. So in this chapter we present some experimental results, that is to say some timings of our system on some common benchmarks.

In the table below, we give the times that it takes to solve some benchmark instances on a standalone RELSAT solver, on a 1-client, 2-client, and 5-client PSolver systems, where the clients were all run on separate machines. All timings are in seconds. The difference between the 1-client PSolver system and the standalone RELSAT solver represents the direct overhead from additional PSolver code. For the 2-client and the 5-client case we also give the speedup achieved. The benchmark instances were obtained from the DIMACS Challenge ftp site (ftp://dimacs.rutgers.edu/pub/challenge/) and from the SAT Group's ftp site (http://sat.inesc.pt/benchmarks/cnf) and are a standard set of benchmark instances originating from model checking, planning, system design, and random problems. We used only unsatisfiable instances, since in the case of satisfiable instances, the results cannot readily be interpreted as it is somewhat a matter of luck how early a solution will be found. All the machines running PSolver software were SunBlade 100 workstations running Solaris under Athena 9.0 (the MIT operating system) in the same cluster. Note: all the runs, including those for standalone RELSAT, were done with the preprocess level for RELSAT set to 0, indicating that no preprocessing is to be done. This is done so that the results are
<table>
<thead>
<tr>
<th>Instance Name</th>
<th>Standalone</th>
<th>1-client</th>
<th>2-Client</th>
<th>Speedup</th>
<th>5-Client</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>barrel5</td>
<td>9</td>
<td>10</td>
<td>5</td>
<td>2.0</td>
<td>5</td>
<td>2.0</td>
</tr>
<tr>
<td>barrel6</td>
<td>105</td>
<td>107</td>
<td>47</td>
<td>2.27</td>
<td>21</td>
<td>5.1</td>
</tr>
<tr>
<td>barrel7</td>
<td>607</td>
<td>602</td>
<td>267</td>
<td>2.25</td>
<td>148</td>
<td>4.06</td>
</tr>
<tr>
<td>barrel8</td>
<td>2311</td>
<td>2315</td>
<td>1043</td>
<td>2.21</td>
<td>423</td>
<td>5.4</td>
</tr>
<tr>
<td>longmult5</td>
<td>8</td>
<td>10</td>
<td>6</td>
<td>1.66</td>
<td>5</td>
<td>2.0</td>
</tr>
<tr>
<td>longmult7</td>
<td>503</td>
<td>505</td>
<td>253</td>
<td>1.99</td>
<td>125</td>
<td>4.04</td>
</tr>
<tr>
<td>longmult9</td>
<td>5092</td>
<td>5094</td>
<td>2646</td>
<td>1.92</td>
<td>902</td>
<td>5.6</td>
</tr>
<tr>
<td>queueinvar10</td>
<td>7</td>
<td>9</td>
<td>6</td>
<td>1.5</td>
<td>6</td>
<td>1.5</td>
</tr>
<tr>
<td>queueinvar14</td>
<td>19</td>
<td>21</td>
<td>11</td>
<td>1.9</td>
<td>6</td>
<td>3.5</td>
</tr>
<tr>
<td>queueinvar18</td>
<td>57</td>
<td>59</td>
<td>31</td>
<td>1.9</td>
<td>18</td>
<td>3.27</td>
</tr>
</tbody>
</table>

Table 8.1: Timings on common benchmarks

Comparable between standalone RELSAT and PSolver, since it would not make sense to run the preprocessing step every time a client is assigned a subproblem. In the future the preprocessing step will be incorporated into the server, but this was not yet done at the time of this experiment.

Unfortunately, we were not able to do more experiments involving a larger number of machines, bigger instances, and a greater variety of instances. This was due to limited resources as far as available disk space and machines go, as well as constraints on the kinds of instances we could use - only unsatisfiable ones, ones that take up some appreciable amount of time to solve, but ones that do not take hours. But even from this limited data, several trends emerge:

- There is some inherent overhead associated with PSolver but it is negligible. Comparing standalone timings to 1-client timings, we see that the latter are 2-3 seconds higher. This difference is constant across instances. Thus, for instances for which PSolver is likely to be useful, this difference is negligible.

- For small instances, the communications and giveup overhead overwhelms parallelization. For instances that took 20 or less seconds to solve, speed-up with 2 clients was usually around 1.5 and increasing clients to 5 had virtually no effect - and could even make the system slower. The reason is that there are several places where increasing clients increases overhead. The communication between server and client are not instant and take time. Each time a client starts solving
a subproblem, some startup cost is incurred. And similarly each time a client
gives up a part of a subproblem, a performance hit is taken. For very small
instances, this overhead outweighs the time gained through parallelization.

- For larger instances, the system performs well. For instances that take a minute
  or more to solve with 1 client, we see speedups that approach and even exceed
  linear.

- The most surprising result was that sometimes speedup exceeded linear. We are
  somewhat at a loss to explain this. We surmise that some factors involved may
  be the varying workload of the machines on which the software was running,
  though we tried to control that as much as possible. Another possible explana-
  tion maybe that during solving a problem, RELSAT creates a lot of supporting
  data structures and is thus quite a memory hog. With more clients, it uses less
  memory as each subproblem it solves is smaller, which may make it faster.

The overall conclusion is that the performance of the system is promising. How-
ever, it is by no means optimized - in fact it is very much not optimized - and further
work in that direction would no doubt improve these numbers. Performance was not
our primary goal from the beginning, and with that in mind, we view these results
as very satisfactory.
Chapter 9

Future Directions

Waiting for the fat lady

While PSolver, in our opinion, represents a big step forward in distributed SAT solving, there is still a lot of work that can be done to improve the system. In this chapter we list some things that we wish we had had time to do. These suggestions range from the more conceptual to the very practical.

9.1 Improving the PSolver protocol

This has two components to it: a more immediate and pragmatic component as well as a more conceptual component. In the immediate future, the protocol needs to be cleaned up. There are some unpleasant inconsistencies in how various pieces of it are defined, which if resolved would make it more conceptually elegant. In addition, it would be desirable for the implementation to have a tool that produces the SendableObject hierarchy directly from the protocol definitions of the data types. Currently this is done by hand, resulting, again in some inconsistencies and an ad hoc approach.

Down the line, there needs to be a revision of the contents of the protocol. Currently there is a somewhat unnecessary distinction between users, servers and clients. Only users can pose SAT problems, only clients can receive subproblems. However, there is not a fundamental difference between user-server interaction and server-client
interaction - ultimately in both cases a problem is posed and a result expected. Therefore, eventually we would like to revise the PSolver protocol to eliminate this distinction. Instead, there will be server/clients, which can both act as servers and as clients. This will help in making the system hierarchical - so that a server can assign a subproblem to a client which, instead of using a sequential SAT solver to solve the problem, will in turn act as a server to its own clients, subdividing the problem further. This poses some new challenges, such as handling giveup requests properly in such client/servers, but will ultimately benefit in creating truly large virtual SAT solving clusters for very large SAT instances.

9.2 Further use of STL

STL is a very well-designed system that, if used consistently throughout a system, can increase its readability and make maintaining it easier. We use STL in several places in PSolver with good results; however, there are still places where using STL instead of the current implementation would improve the system. In particular one such place is the CommunicationObject hierarchy. It makes sense to change this hierarchy to conform to STL’s iostreams library. Doing so would enable us to easily make use of other systems which use the iostreams paradigm, and to more easily abstract away from the particular mechanisms used to accomplish the physical transfer of bytes between processes. It would make easier, for example, to add encryption and compression to the current communications system.

9.3 Making more use of information from clients

One very important future direction is exploring how information gained during solving a subproblem in one client can be used to help another client in solving a different subproblem. One approach might be to share conflict or no-good information between clients, as a no-good obtained from one subproblem may well be applicable in another subproblem. On an even simpler level, sometimes particular variable assign-
ments can be deduced in one client and then shared with other clients. This is the most important direction as far as getting more speed-up out of parallelization.

### 9.4 Incorporating the progress bar

In a separate paper we describe a progress bar we have developed for SAT solvers. This progress bar gives an approximation, using several heuristics, of how far along a SAT solver is in its solution of a problem. This can be very useful in a distributed SAT system. One way it can be used, for instance, is by running several different clients, with different sequential SAT solving algorithms, on the same problem for a short period of time, then using the progress bar to determine which one is doing the best job, and stopping the other ones. Because different algorithms can perform very differently on particular types of SAT instances, this can improve solution times a lot. Also it would be useful to report to the user how much longer solving the problem is expected to take. Adapting the progress bar to PSolver requires some modifications, since it must be able to aggregate across several subproblems, but it should not be too difficult.

### 9.5 A Java client

There are several SAT solving algorithms that have Java implementations. It would therefore be useful to write a Java implementation of the client component of our system.

### 9.6 A web interface

It may well be very useful to create a web front-end for communicating with the PSolver system - it would probably lead to more people using it and testing it.
Chapter 10

Conclusion

*Enough of this already*

In the end, we think PSolver is a worthy achievement. While it has several faults and has much room to improve, it makes an important contribution to the SAT solving community, as it is the first distributed solver to have a formally defined structure that allows it to be extensible and make use of several different SAT solving algorithms. It is also the only distributed SAT solver which is truly scalable and truly requires only general purpose hardware. And, in its current incarnation, it has already been shown to produce good performance.

Our group is currently working on a new version of our object modeling tool, alloy. We intend this new version to use PSolver as its backend, and we expect that PSolver will contribute to the tool being a success.
Appendix A

The PSolver protocol

Like candy from a baby

We’ve earlier discussed the semantics of the PSolver protocol. Here we discuss the syntax - exactly how do messages in PSolver protocol look.

In general, a PSolver message consists of a message name, followed by a list of named arguments. The list of arguments is variable-length, and depends on the message names. In general it consists of a list of pairs, each pair being an argument name and an argument value. An argument value can be a complex data type.

Below we give the grammar for a PSolver Protocol message.

```
PSOLV_MESSAGE = BEGINSEQUENCE MESSAGE ENDSEQUENCE
BEGINSEQUENCE = '<<<'
ENDELEMENT = '>>>'
MESSAGE = '<' message_name ARGUMENTLIST '>
ARGUMENTLIST = ε | ARGUMENTPAIR ARGUMENTLIST
ARGUMENTPAIR = '<' arg_name arg_value '>
```

Figure A-1: Psolver Protocol message grammar

Next, we give the table of message names and the corresponding list of arguments. For each message name, the list of arguments consists of argument names and types. We also give the meaning of the argument, if necessary.
<table>
<thead>
<tr>
<th>Message Name</th>
<th>Argument Name</th>
<th>Argument Type</th>
<th>Argument Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>user.solve</td>
<td>default</td>
<td>SATUserProblem</td>
<td></td>
</tr>
<tr>
<td>user.resume</td>
<td>problemId</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>user.cancel</td>
<td>clientId</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>user.next</td>
<td>clientId</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>user.done</td>
<td>clientId</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>user.quit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>server.user.ack</td>
<td>problemId</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>clientId</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>server.result</td>
<td>clientId</td>
<td>int</td>
<td>the result</td>
</tr>
<tr>
<td></td>
<td>default</td>
<td>SATUserResult</td>
<td></td>
</tr>
<tr>
<td>Message Name</td>
<td>Argument Name</td>
<td>Argument Type</td>
<td>Argument Meaning</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------</td>
<td>---------------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>server.ack</td>
<td>clientId</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>server.problem.solve</td>
<td>clientId</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td></td>
<td>problemId</td>
<td>string</td>
<td>the subproblem</td>
</tr>
<tr>
<td>server.problem.solve</td>
<td>default</td>
<td>SATProblem</td>
<td></td>
</tr>
<tr>
<td>server.problem.giveup</td>
<td>clientId</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td></td>
<td>problemId</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>server.problem.cancel</td>
<td>clientId</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td></td>
<td>problemId</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>server.quit</td>
<td>clientId</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td></td>
<td>problemId</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>server.file.send</td>
<td>clientId</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td></td>
<td>problemId</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>default</td>
<td>SATClauseList</td>
<td>the instance</td>
</tr>
<tr>
<td>client.new</td>
<td>default</td>
<td>SATClientInfo</td>
<td>client capabilities</td>
</tr>
<tr>
<td>client.result</td>
<td>clientId</td>
<td>int</td>
<td>the result</td>
</tr>
<tr>
<td></td>
<td>problemId</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>default</td>
<td>SATResult</td>
<td></td>
</tr>
<tr>
<td>client.progress</td>
<td>clientId</td>
<td>int</td>
<td>guiding path describing subproblem left to solve</td>
</tr>
<tr>
<td></td>
<td>problemId</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>default</td>
<td>SATGuidingPath</td>
<td></td>
</tr>
<tr>
<td>client.status</td>
<td>clientId</td>
<td>int</td>
<td>the status message</td>
</tr>
<tr>
<td></td>
<td>problemId</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>default</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>Message Name</td>
<td>Argument Name</td>
<td>Argument Type</td>
<td>Argument Meaning</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------</td>
<td>---------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>client.giveup</td>
<td>clientId</td>
<td>int</td>
<td>the subproblem being given up</td>
</tr>
<tr>
<td></td>
<td>problemId</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>giveupPath</td>
<td>SATGuidingPath</td>
<td>the subproblem the client is left with</td>
</tr>
<tr>
<td></td>
<td>remainingPath</td>
<td>SATGuidingPath</td>
<td></td>
</tr>
<tr>
<td>client.giveup_refuse</td>
<td>clientId</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td></td>
<td>problemId</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>client.file_request</td>
<td>clientId</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td></td>
<td>problemId</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>

**Monitor subprotocol**

<table>
<thead>
<tr>
<th>Message Name</th>
<th>Argument Name</th>
<th>Argument Type</th>
<th>Argument Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>server.register</td>
<td>default</td>
<td>ServerInfo</td>
<td>the server location</td>
</tr>
<tr>
<td>server.dead</td>
<td>default</td>
<td>ServerInfo</td>
<td>server location</td>
</tr>
<tr>
<td>client.server.request</td>
<td>clientId</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td></td>
<td>default</td>
<td>ServerInfo</td>
<td>a server location</td>
</tr>
<tr>
<td>monitor.server</td>
<td>clientId</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>monitor.server_none</td>
<td>clientId</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>monitor.server_list</td>
<td>clientId</td>
<td>int</td>
<td>list of server locations</td>
</tr>
<tr>
<td></td>
<td>default</td>
<td>ServerInfoList</td>
<td></td>
</tr>
</tbody>
</table>

Next we describe how datatypes are encoded. There are four intrinsic datatypes, bool, int, short-string and long-string, the special constructs enum and list<datatype> and complex datatypes. The intrinsic datatypes are encoded as follows:

- **int** is encoded as simply the ascii representation of the number.
- **int** is encoded as 1 for true, and 0 for false.
- **short-string** is encoded as simply the value of the string. It cannot include spaces or the characters < and >.
The special construct `enum` represents an enumeration - a list of named constants. It is encoded as an `int` with the value of the `int` being the index of the constant in the enumeration. For `enum` fields, the list of constants is given in the field meaning column.

The special construct `list<datatype>` represents a list of values of another datatype. It is encoded simply as `value - 1value - 2..value - n`, with each value encoded appropriately.

Complex datatypes are basically structs consisting of several fields, which in turn could be intrinsic datatypes or complex datatypes. Complex datatypes are encoded as `< field - 1field - 2...field - n >`, with the fields being separated by spaces. The order of the fields matters. The following tables gives the complex datatypes and their definitions. For each datatype, it lists what the datatype is supposed to represent, the field types and the meaning of the field. The order in which the fields are listed is the order in which they must be encoded.
<table>
<thead>
<tr>
<th>Datatype Name</th>
<th>Datatype meaning</th>
<th>Field type</th>
<th>Field Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SATLiteralSplit</td>
<td>a literal-split pair in a guiding path</td>
<td>int</td>
<td>the literal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bool</td>
<td>whether this is an open or closed pair (true for open)</td>
</tr>
<tr>
<td>SATGuidingPath</td>
<td>a guiding path</td>
<td>int</td>
<td>the number of splits in the guiding path</td>
</tr>
<tr>
<td></td>
<td></td>
<td>list&lt;SATLiteralSplit&gt;</td>
<td>the list of literal splits. The order of the splits matters.</td>
</tr>
<tr>
<td>SATClause</td>
<td>a clause in a SAT instance - a list of literals</td>
<td>int</td>
<td>number of literals in the clause</td>
</tr>
<tr>
<td></td>
<td></td>
<td>list&lt;int&gt;</td>
<td>the list of literals in the clause</td>
</tr>
<tr>
<td>Datatype Name</td>
<td>Datatype meaning</td>
<td>Field type</td>
<td>Field Meaning</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------</td>
<td>------------</td>
<td>---------------</td>
</tr>
<tr>
<td>SATClauseList</td>
<td>a SAT instance expanded to include dependency information</td>
<td>int</td>
<td>the number of clauses in the instance</td>
</tr>
<tr>
<td></td>
<td>list&lt;SATClause&gt;</td>
<td>int</td>
<td>the list of clauses</td>
</tr>
<tr>
<td></td>
<td>bool</td>
<td>the number of variables in the clause</td>
<td></td>
</tr>
<tr>
<td></td>
<td>list&lt;int&gt;</td>
<td>bool</td>
<td>whether dependency information is available. If it is not, all variables are considered independent</td>
</tr>
<tr>
<td></td>
<td>list&lt;int&gt;</td>
<td>list&lt;int&gt;</td>
<td>the list of independent variables. The field is only present if the value of the previous field is true</td>
</tr>
<tr>
<td>Datatype Name</td>
<td>Datatype meaning</td>
<td>Field type</td>
<td>Field Meaning</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------</td>
<td>------------</td>
<td>---------------</td>
</tr>
<tr>
<td>SATProblem</td>
<td>a subproblem as assigned by a server to a client</td>
<td>SATGuidingPath</td>
<td>the guiding path representing the subproblem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>short-string</td>
<td>the filename for the file containing the SAT instance for the problem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>enum</td>
<td>the representation for the SAT instance used in the file. The enumeration is { SAT, CNF }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int</td>
<td>timeout interval in seconds - how many seconds the client should solve the subproblem for before giving up</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int</td>
<td>progress interval in seconds - the number of seconds the client should wait before sending progress updates</td>
</tr>
<tr>
<td>Datatype Name</td>
<td>Datatype meaning</td>
<td>Field type</td>
<td>Field Meaning</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------</td>
<td>------------</td>
<td>---------------</td>
</tr>
<tr>
<td>SATResult</td>
<td>the result of solving a subproblem as reported by a client to the server</td>
<td>SATGuidingPath</td>
<td>the value of this field depends on the type of result returned (as indicated by the next field). If the value is UNSAT or ERROR, this field is not used and its value can be anything. If the value is SAT the guiding path should represent the satisfying assignment as well as the subproblem left to solve when it was found. For TIMEOUT or CANCELLED or QUIT or PROGRESS, the guiding path should represent the subproblem left to solve when the timeout period expired or the cancel or quit message was received, or progress report was made. An enumeration of constants representing the result. The enumeration is SAT, UNSAT, TIMEOUT, CANCELLED, PROGRESS, QUIT, ERROR. SAT: a satisfying assignment was found, UNSAT: no satisfying assignment, TIMEOUT: timeout period expired, PROGRESS: this is a progress report, CANCELLED: a cancel message was received, QUIT: a quit message was received, ERROR: an error occurred.</td>
</tr>
<tr>
<td>Datatype Name</td>
<td>Datatype meaning</td>
<td>Field type</td>
<td>Field Meaning</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------------------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SATUserProblem</td>
<td>a problem as assigned by a user to a client</td>
<td>short-string</td>
<td>the filename of the file containing the SAT instance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>enum</td>
<td>the representation for the SAT instance used in the file. The enumeration is SAT, CNF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>short-string</td>
<td>the filename of the file into which to write the satisfying assignment, if one is found.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int</td>
<td>timeout interval (in seconds) - how long should the server work on the problem before giving up</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bool</td>
<td>whether the server should save the state of its computation for possible future resuming for this problem</td>
</tr>
<tr>
<td>Datatype Name</td>
<td>Datatype meaning</td>
<td>Field type</td>
<td>Field Meaning</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------------------------------------------------------------------</td>
<td>------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SATUserResult</td>
<td>the result of solving a problem as reported by the server to the user</td>
<td>enum</td>
<td>the result. The enumeration is SAT, UNSAT, TIME-OUT, CANCELLED, QUIT, ERROR. SAT means a satisfying assignment was found, UNSAT means a satisfying assignment does not exist, TIME-OUT means the timeout period expired without finding a satisfying assignment or finishing solving the problem, CANCELLED means a cancel message was received, QUIT means a quit message was received, and ERROR means an error occurred.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long-string</td>
<td>this field is only used if the value of the previous field is ERROR. In this case this field contains the string describing the error.</td>
</tr>
</tbody>
</table>
Appendix B

Instrumenting Solvers

I've already got one!

1. You can write a client that implements the PSolver protocol. This means the client must be able to establish a connection to a port on a given machine, and must be able to send and receive ascii data. At the minimum the client should be able to implement the client-server subprotocol of the PSolver protocol. If the client wishes to take advantage of the monitor component, then it must be able to implement the monitor subprotocol as well. The PSolver protocol semantics are described in Chapter 6 and its syntax is given in Appendix B. This method is necessary if the solver is written in a language other than C/C++, or if its algorithm is such that instrumenting it for our existing client as described below would be for whatever reason infeasible.

2. If the solver is written in C/C++ you can save the effort of writing a client from scratch and use our client. In this case all that needs to be done is:

   - write a solver stub class that is a subclass of the SolverStub. This class must implement the methods solve and getClientInfo. The solve method takes three arguments: the input stream containing the SAT instance, the guiding path representing the subproblem, and a solver control object which must be queried during solving for important events occurring. It must return the result of solving the subproblem. The
getClientInfo method must return a ClientInfo object describing the capabilities of this particular solver.

- The solver must periodically query the solver control object passed to the solve method. In particular, the solver must invoke the process method which processes any events that occurred since the last query. Then, the solver must invoke several methods which check for whether particular events, such as receiving cancel or quit messages, occurred. If any of these events occurred, the solver must then respond accordingly. In particular, here are the methods to check for the particular events and how the solver must respond to them:

  - shouldCancel - returns true if a cancel message has been received. If a cancel message has been received, the solver must stop solving the subproblem it is working on. The solve method must return with a CANCELLED result.

  - shouldQuit - returns true if a quit message has been received. If a quit message has been received, the solver must stop solving the subproblem it is working on. The solve method must return with a QUIT result.

  - shouldTimeout - returns true if the timeout period has expired. If the timeout period has expired, the solver must stop solving the subproblem it is working on. The solve method must return with a TIMEOUT result.

  - shouldProgress - returns true if it is time for a progress report. The solver should invoke the progress method of the solver control object with the guiding path representing the subproblem the solver has left to solve.

  - shouldGiveup - returns true if a giveup message has been received. The solver must either invoke the refuseGiveup method of the solver control object to indicate it does not wish to fulfill the giveup request,
or the \texttt{giveup} method with the guiding path representing the subproblem being given up and the new subproblem it has left to solve. The solver must adjust its internal data structures to reflect the new subproblem it is working on.

That is all! All that remains is to modify the Makefile to include any new object files that need to be linked in (these includes the stub class and the solver object files) and then make the satclient executable. Please refer to the source code, Appendix A, and Appendix D for more information.
Appendix C

PSolver user guide

I want to play with my new toy

The following is the user manual that comes with the software distribution of PSolver. It describes how to set up and use the PSolver distribution.

The PSolver webpage is http://sdg.lcs.mit.edu/satsolvers/PSolver/index.html. This page will have the most up-to-date binary distribution, and (soon) the source code.

The PSolver distribution consists of 4 executables, a configuration file, and a text version of this document. They are extracted into the subdirectory called PSolver underneath the directory into which you downloaded the zip file. The four executables are the four pieces of the PSolver system. They are:

- **satserver** - the server piece.
- **satclient** - the client piece.
- **satmonitor** - the monitor piece.
- **satuser** - the user interface.

All four executables read certain configuration options from the configuration file solvers.cfg. The configuration file has sections for each of the executables.

There are two patterns of using the PSolver system - with and without the monitor piece. The monitor piece is useful if you want to establish a long-term cluster for SAT
solving. If all you want is to solve one or two instances using a few machines, then you can omit the monitor and have the clients connect directly to the server.

C.1 Setting up and running the system

Below we describe the basic procedure for setting up and using the system, both in the case where a monitor is used and where it is not.

1. With the monitor

   • setup

      (a) place the satserver executable and a copy of solvers.cfg on the machine on which the server will be run.

      (b) place the satmonitor executable and a copy of solvers.cfg on the machine on which the monitor will be run. This can be same as server; in general this should be a machine that almost never goes down.

      (c) place the satclient executable and a copy of solvers.cfg on all machines which you wish to act as clients. This should be most machines on the network.

      (d) place the satuser executable and a copy of solvers.cfg on any machine from which you’d like to be able to use the system. This can be same as a server or a client machine. It is helpful if the machine shares file namespace with the server machine.

      (e) run the satmonitor executable. The convention is to run it on port 5001.

      (f) run the satclient executables on all client machines, pointing them at the satmonitor.

   • running

      (a) run the satserver executable, pointing it at the satmonitor. The convention is to run it on port 5000.
2. Without the monitor.

- setup
  
  (a) place the satserver executable and a copy of solvers.cfg on the machine on which the server will be run.

  (b) place the satclient executable and a copy of solvers.cfg on all machines which you wish to act as clients. This should be most machines on the network.

  (c) place the satuser executable and a copy of solvers.cfg on any machine from which you’d like to be able to use the system. This can be same as a server or a client machine. It is helpful if the machine shares file namespace with the server machine.

- running

  (a) run the satserver executable. The convention is to run it on port 5000.

  (b) run all satclient executables on all client machines and point them at the satserver.

  (c) run satuser, pointing it at the satserver. Use satuser to give SAT problems to the PSolver system and receive solutions. The satuser interface is explained below.

  (d) repeat b as necessary. multiple problems can be posed to satserver simultaneously by running multiple copies of satuser (on same or
different machines).

(e) if satserver crashes, repeat from step a.

To be concrete, we describe a specific scenario for a particular network configuration and give the exact commands that should be run on each machine. For the meaning of the various command-line parameters, refer to the following sections where we describe in detail each executable and its command-line parameters and configuration-file options.

C.2 A Concrete Example

Here we describe an example use of the system. We assume a network of 10 computers, with hostnames m1, m2, .. m10. We assume they can all write and read the directory /gl/sat-files. This directory contains some SAT problems in the cnf format, with filenames problem1.cnf and problem2.cnf. We assume that the PSolver executables have been put on the appropriate machines as described above. We indicate on which machine the below commands are typed by using the machine name as the prompt.

1. Example using the monitor piece. We run the monitor and the server on machine m1, clients on machines m2-m10, and the satuser from machine m4.

[setup phase]

m1>satmonitor 5001 solvers.cfg
m2>satclient relsat solvers.cfg -monitor m1 5001
m3>satclient relsat solvers.cfg -monitor m1 5001
.. 
m10>satclient relsat solvers.cfg -monitor m1 5001

[solve the problem problem1.cnf]

m1>satserver 5000 solvers.cfg -register m1 5001
m4>satuser solvers.cfg -monitor m1 5001
at satuser’s prompt
solve /g1/sat-files/problem1.cnf cnf /g1/sat-files/problem1.sol 10000 1
- satuser tells us problemId is prob10001_1
... assume server crashes, we want to resume solving

m1>satserver 5000 solvers.cfg -register m1 5001
m4>satuser solvers.cfg -monitor m1 5001
    at satuser’s prompt
    resume prob10001_1

2. Example without using the monitor piece We run the server on machine m1,
clients on machine m2-m10, and satuser from machine m4

[solve the problem problem1.cnf]
m1>satserver 5000 solvers.cfg
m2>satclient relsat solvers.cfg m1 5000
m3>satclient relsat solvers.cfg m1 5000
  ...
m10>satclient relsat solvers.cfg m1 5000
m4>satuser solvers.cfg m1 5000
    - at satuser’s prompt
    solve /g1/sat-files/problem1.cnf cnf /g1/sat-files/problem1.sol 10000 1
    - satuser tells us problemId is prob10001_1

  .. if server crashes, we want to resume solving
m1>satserver 5000 solvers.cfg
m2>satclient relsat solvers.cfg m1 5000
m3>satclient relsat solvers.cfg m1 5000
  ...
m10>satclient relsat solvers.cfg m1 5000
m4>satuser solvers.cfg m1 5000
C.3 Command Reference

Below we describe each executable and explain the command line parameters and configuration options it accepts.

C.3.1 SATMONITOR

Command line: satmonitor port-number config-file

The satmonitor executable acts as a clearinghouse of information about the satservers running on the network. It allows satclient and satuser to locate a satserver to connect to. It is not necessary to run this application - you can just run a satserver and tell satclients and satusers its location directly. However, if you intend to set up a fairly stable PSolver network, where satservers can go down (because of program crashes or other reasons) the satmonitor makes it much more convenient to allow satclients and satusers to reconnect to the satserver without human intervention.

Command line parameters:

- **port-number** - the port number on which satmonitor will listen for connections.
- **config-file** - the name of the configuration file where satmonitor looks for configuration parameters.

Configuration file parameters (see also common configuration parameters): none except for the common ones

C.3.2 SATSERVER

Command line: satserver port-number config-file [-register
satserver is the server piece of the distributed solver. This is the application that manages clients, assigns them subproblems, keeps track of the subproblems and returns solutions to the user. You do not give problems to satserver directly, however; the satuser application is used for that (alternatively one can use the PSOLV protocol to communicate with the satserver directly by connecting to the appropriate port; satuser is just an application to hide that from the end-user).

There are two modes in which you can run satserver. The first is stand-alone. In this case only the first two command line parameters are used, and the satserver is not registered with a satmonitor. Therefore, satclients and satusers must be given the address and port number of the satserver directly. In the second mode the satserver is registered with a satmonitor. In this case all 5 command-line parameters are used.

Command-line parameters:

- **port-number** - the port number on which satserver will listen for connections.

- **config-file** - the name of the configuration file where satserver looks for configuration parameters.

- **-register** - this must be the 3rd command-line parameter if the satserver is running in registered mode.

- **monitor-hostname** - hostname of the machine on which the satmonitor application with which this satserver will register itself is running.

- **monitor-port-number** - the port number on which the satmonitor application is running.

Configuration file parameters (see also common parameters):

- **progressInterval** - the interval between progress messages from satclients, in seconds. The longer this is, the less often satserver receives updates from its clients. This makes it more efficient, but if a client crash more information is lost. The default value is 60 seconds.
• **saveInterval** - the interval between saves of server state for a particular problem. If the save state option is enabled for a particular problem, this is the interval between saves of the state of the server for that problem. Saving state enables the server to recover and continue solving the problem from where it left off if the server crashes. Again, longer intervals result in greater efficiency but more lost information. The default value is 120 seconds.

• **pathComparatorType** - this chooses the heuristic used to do load balancing. There are currently two heuristics: psato and dk. In different cases one may be better than the other. We currently don’t have a great idea of which one is better when, so you can change this if you wish or leave it alone. The default is psato.

### C.3.3 SATCLIENT

Command line: `satclient solver-name config-file [-monitor]`

**satclient** is the client piece of the distributed solver. It is run on every computer that wishes to be part of the solver network. It is the part that actually solves SAT problems. Note that we did not write our own single-process SAT solver. Instead, the satclient application is a framework into which various existing SAT solvers can be plugged in by instrumenting them accordingly. Currently this has been done for one solver - RELSAT by Roberto Bayardo.

**satclient** can be run in two modes. In the direct connection mode, the **satclient** is given directly the hostname and portnumber of a **satserver** and connects directly to that **satserver**. If the **satserver** goes down, the **satclient** stops running. In the monitored mode, the **satclient** is given a **satmonitor** hostname and portnumber and uses the **satmonitor** to locate a **satserver** to connect to. If that **satserver** goes down, the **satclient** will query **satmonitor** for another. If no **satservers** are available, the **satclient** will continue running and periodically query **satmonitor** for a **satserver**. The monitored mode is selected by using -monitor as the 3rd command-line parameter.
Command line parameters:

- **solver-name**: the name of the SAT solver to use. As mentioned above the satclient application is a framework into which various SAT solvers can be plugged in and used. This parameter specifies the name of the SAT solver to use for this satclient. Currently the only possible value for this is relsat.

- **config-file**: the name of the configuration file where satclient looks for configuration parameters.

- **-monitor**: this is used to select the monitored mode. If this parameter is omitted, then the direct connection mode is used.

- **hostname**: if direct connection mode, hostname on which satserver is running. If monitored mode, hostname on which satmonitor is running.

- **port-number**: if direct connection mode, port number on which satserver is running. If monitored mode, port-number on which satmonitor is running.

Configuration file parameters (see also common parameters):

- **GiveupDelay**: this is a fairly technical parameter and should for the most part be left alone.

- **connectTimeout**: this only has meaning in monitored mode. In monitored mode, it is the time between successive tries to obtain a satserver from satmonitor and to connect to it if one is available.

**C.3.4 SATUSER**

Command line: satuser config-file [-monitor] hostname port

satuser is the interface to the psolver system for the end-user. It is made available to make interaction with psolver easier. Instead of using satuser, one can also connect directly to a satserver on the appropriate hostname and port and use the PSOLV protocol to communicate. This is what a machine user of PSOLV (i.e. another
application) would do. But for a person who wants to use psolver directly, satuser provides the easiest interface.

Satuser can be run in two modes: direct connection and monitor. In direct connection mode the hostname and port of the satserver is specified directly. In monitor mode, satuser obtains this information from a satmonitor. The monitor mode is specified by using the -monitor command-line parameter.

Once satuser runs and connects to satserver, it prompts for certain commands. The interaction with satuser will be explained in a section below.

Command line parameters:

- **config-file** - the name of the configuration file where satuser looks for configuration parameters.

- **-monitor** - this is used to select the monitor mode. If this parameter is omitted, then the direct connection mode is used.

- **hostname** - if direct connection mode, hostname on which satserver is running. If monitored mode, hostname on which satmonitor is running.

- **port-number** - if direct connection mode, port number on which satserver is running. If monitored mode, port-number on which satmonitor is running.

Configuration file parameters (see also common parameters): none except for common ones

### C.3.5 COMMON CONFIGURATION FILE PARAMETERS

Some configuration file parameters are used by all 4 applications. Each application has its own copy of the parameters (in its section), but the meaning of the parameters is the same for each application, so they are explained below:

- **multithreaded** - all psolver applications can be run in multithreaded or singlethreaded mode. The behavior is the same regardless. This is more of a developed parameter and its value should not be changed from its default (0,
meaning single-threaded mode), unless you're curious. Note: if running under
windows, multithreaded mode cannot be used, or all executables will crash.

- **logfile** - this is the name of the file to which to log messages produced by
  the executable. All psolver executables print messages in response to certain
  events. Mostly these are used for debugging purposes. The messages are of four
  types according to the severity of the event - STATUS, WARNING, ERROR
  and FATAL ERROR. If desired these messages can be logged to a file. This
  specifies the name of the file. If you do not wish to log them, leave this parameter
  blank. In general logging should be done so that if crashes occur you can send
  us the log files.

- **displayStatusMessages** - this parameter controls whether STATUS type mes-
  sages should be printed to the screen. STATUS messages are generally debug-
  ging output during normal operation. They can be quite numerous and slow
  down the applications significantly, so for the most part the value of this should
  be 0, meaning do not print them, unless you're actually trying to follow what
  is happening under the hood.

- **logStatusMessages** - whether STATUS type messages should be logged to the
  logfile, if one is being used. In general this should be 1, meaning they should
  be logged.

### C.3.6 Using SATUSER

Here we describe how to use satuser to interact with the psolver system. For the
most part its prompts are fairly self explanatory.

After successfully connecting to the satserver, satuser prompts you to enter one
of three commands: solve, resume, or quit. quit immediately quits the application.
The other two take some argument and are further documented below.
SOLVE

format: solve filename format solution_filename timeout save_state

This asks the psolver system to solve a SAT problem. The parameters are:

- **filename** - the name of the file containing the SAT problem. Note this name must be readable, as given, by the **satserver** executable to which this **satuser** is connected. This, granted, is somewhat inconvenient and will be remedied in the next release. But for now that is the case. It actually does not need to be readable by the **satuser** executable.

- **format** - the format in which the SAT problem is encoded. Currently the only format supported is the cnf format as defined by the DIMACS challenge.

- **solution_filename** - the name of the file to which a solution, if one exists should be written. This name must be writeable by **satserver** and readable by **satuser**.

- **timeout** - how long psolver should try solving the problem before giving up.

- **save_state** - this should be 1 if we want to be able to recover from a **satserver** crash or resume after cancelling. Otherwise, 0.

Once the solve command is entered, **satuser** will send it to **satserver**. **satserver** should acknowledge it and return to **satuser** a problemId. This problemId can be used later to resume solving the problem if the **satserver** crashes.

RESUME

format: resume problemId

this resumes a solving a problem that was interrupted (either by a cancel command or by a server crash). Parameters are:

- **problemId** - the problemId for the problem which to resume. This is the problemId which **satuser** prints out after a solve command.
**DURING SOLVING**

While the problem is being solved, satuser presents a prompt to either cancel or quit. Cancel stops solving the problem and returns to the solve/resume/quit prompt. It can later be resumed if save_state option was 1. Quit stops solving the problem and terminates the satuser executable.

**AFTER SOLVING**

When psolver finishes solving the problem, it returns the result to satuser which displays it. The result is either UNSAT, meaning the problem is unsatisfiable, or SAT, meaning there is a solution. In the latter case the solution is also displayed. If the result is UNSAT, then satuser returns immediately to the solve/resume/quit prompt. If the result is SAT, however, satuser prompts either for the next command or the done command. The next command, requests another solution. If there are no more solutions, the UNSAT result will be returned. The done command returns to the solve/resume/quit prompt.
Bibliography


