

### **Abstract**

We present CRISP, an extensible efficient wire protocol for incremental SAT solving. The protocol allows applications to communicate with a remote or local server over tcp or unix domain sockets. It efficiently transfers data and minimizes unnecessary round trips. It is simple, and easily extensible to domains such as optimisation, quantifiers, proofs, etc. An open source reference implementation is available and discussion for future versions and extensions takes place in public.

# CRISP: The CompRessed Incremental SAT Protocol

Scott Cotton, IRI France, SAS

July 7, 2016

## 1 Introduction

SAT solvers are traditionally linked against applications, and very often the application context yields a high ratio of very easy problems to medium or hard problems. "Big gun" SAT solvers are often NOT used in applications because of the code complexity and/or hardware requirements. Additionally, SAT solvers are usually written in C or C++, whereas higher level application contexts may involve a software stack with many languages and components. The inter-language bindings create integration and computational overhead. To address these issues and at once provide a useful piece of infrastructure for cloud and distributed deployment, we designed CRISP, the CompRessed Incremental SAT Protocol.

The base protocol can be seen as a wire protocol client-server version of the standard incremental SAT solving interface originally defined in [1].

The rest of this paper is organised as follows. Section 2 details the wire protocol. Section 3 defines the generic extension mechanism and two useful extensions (optimisation and assumption based multi-plexing). Section 4 gives some experimental data. Section 5 concludes.

## 2 The Protocol

### 2.1 Wire Data

Every piece of data communicated between the client and the server takes the logical form of a uint32, an unsigned 32 bit value.

This space houses literals and communication instructions/directives between the client and the server. These instructions/directives are called protocol points in the following.

We only use 17 protocol points, but to enable extensions, we reserve 256 integers for protocol points, at the high end of the range representable by uint32. The rest of the space is used to house variables and literals, coded in the tradition of SAT solvers (see Adding below).

## 2.2 Flow Overview

Here we present an overview of the interactions in CRISP.

1. Client negotiates connection with server
2. Client then requests (`<add>` or `<assume>`) as many times as it likes. The Server does not respond to these requests.
3. Client then requests `<solve>`. This enters a loop between the client and the server on the same connection as follows:
  - (a) Client: `<solve>`
  - (b) Server: `<unknown>|<sat>|<unsat>|<end>`
  - (c) Client: `<continue>|<end>`
4. Client sends (`<model>` or `<modelfor>` or `<failed>` or `<failedfor>`) as many times as it likes. Each time it sends one of these operations, the server responds with the requested data.
5. Optionally, the client may send `<reset>`. In this case, the server forgets added clauses and the flow goes to item 2 above.
6. Client sends `<quit>`, both ends disconnect

In the above, step 3,

(b,c) repeats until server sends `<sat>`, `<unsat>`, or `<end>`

Thus step 3 can be represented by the regular expression below, where vertical alignment represents concatenation.

client: <code>&lt;solve&gt;</code> (server: <code>&lt;unknown&gt;</code> client: <code>&lt;continue&gt;</code>  client: <code>&lt;end&gt;</code> )* (server: <code>&lt;end&gt;</code>   <code>&lt;sat&gt;</code>   <code>&lt;unsat&gt;</code> )
---

The meaning of the server sending `<unknown>` is that it doesn't know the answer and is willing and able to make more progress solving the problem. The meaning of the server initiating `<end>` (in response to client `<solve>` or client `<continue>`) that it is unable or unwilling to make more progress on solving the problem and does not know the answer.

In this loop, the client does not need to wait between reads and writes, since presumably the server is doing the solving. On the other hand, the server needs not try to respond so fast as to eat up unnecessary resources. So the server may do this at an appropriate frequency.

If we call the regular expression above S, the allowable overall (error-free) flow interactions can be represented by the regular expression

$$1((2^*S4^*)^*5?)^*6$$

There are also error conditions, corresponding to further constraints on the allowable flows; these simply abort the sequence and cause a disconnect; these are not represented in this overview but rather are detailed below.

## 2.3 Varuint Encoding

The wire format then uses var-uint encoding (a stream of bytes each of which indicates whether it is the last with 1 bit and uses 7bits to encode the non-zero LSBs of the value). This is a standard variable width encoding which compresses values which tend to be small. This works as follows.

Let  $u_i$  for  $i$  in  $0..31$  be the bits of an unsigned 32 bit int  $u$ . The encoding of  $u$  is a slice of bytes  $e = e_0e_1 \dots$

$e_0$  houses the 7 most LSB of  $u$ : here denoted  $u_{0..7}$

If  $u \gg 7 = 0$ , then the length of  $e$  is 1.

Otherwise  $e_1$  houses  $u_{7..14}$ .

If  $u \gg 14 = 0$ , then the length of  $e$  is 2.

Otherwise  $e_2$  houses  $u_{14..21}$ .

...

This continues for all 32 bits. Also, For every element  $e_i$  of  $e$  except the last,  $e_i \neq 0$ .

The server may decode a stream of varuint32 data and then, for each value  $v$ , test whether it is a command/code point by testing

$$v \geq (2^{32} - 1) - 255$$

if true,  $v$  is a command, if false,  $v$  is a literal.

### 2.3.1 Rational

We distinguish the special value '0' as a literal to zero-terminate clauses and lists of assumptions or lists of failed literals. This is conventional and only requires 1 byte in varuint format. Commands happen with much less frequency in the protocol when there is a potential need to send lots of data fast (eg loading a big dimacs). Commands are large values and hence have larger varuint size.

## 2.4 Protocol Points

We reserve 256 code points for extensions to learning, optimisation, etc. In this proposal we have only the following op codes

$\langle \text{add} \rangle$	$(2^{32} - 1)$
$\langle \text{assume} \rangle$	$(2^{32} - 1) - 1$
$\langle \text{solve} \rangle$	$(2^{32} - 1) - 2$
$\langle \text{continue} \rangle$	$(2^{32} - 1) - 3$
$\langle \text{end} \rangle$	$(2^{32} - 1) - 4$
$\langle \text{error} \rangle$	$(2^{32} - 1) - 5$
$\langle \text{failed} \rangle$	$(2^{32} - 1) - 6$
$\langle \text{failedfor} \rangle$	$(2^{32} - 1) - 7$
$\langle \text{model} \rangle$	$(2^{32} - 1) - 8$
$\langle \text{modelfor} \rangle$	$(2^{32} - 1) - 9$
$\langle \text{sat} \rangle$	$(2^{32} - 1) - 10$
$\langle \text{unsat} \rangle$	$(2^{32} - 1) - 11$
$\langle \text{unknown} \rangle$	$(2^{32} - 1) - 12$
$\langle \text{quit} \rangle$	$(2^{32} - 1) - 13$
$\langle \text{reset} \rangle$	$(2^{32} - 1) - 14$
$\langle \text{key} \rangle$	$(2^{32} - 1) - 15$
$\langle \text{ext} \rangle$	$(2^{32} - 1) - 16$

The remaining protocol points are for specific extensions.

## 2.5 Variables and Literals

Variables are Boolean (can be either true/false) and each variable is associated with a uint32. A "literal" in propositional logic is just a variable or the logical negation of a variable. CRISP uses standard SAT-solver encoding of variables and literals.

If a variable is indicated by some uint32, say  $x$ , then a literal  $m$  over  $x$  is

$$\begin{aligned} x \ll 1 & \quad \text{if } m \text{ is positive} \\ (x \ll 1) | 1 & \quad \text{if } m \text{ is negative} \end{aligned}$$

Since we have uint32 coding, and 256 coding points, the maximum variable representable in the protocol is

$$((2^{32} - 1) - 256) \gg 1$$

or equivalently

$$2147483519$$

## 2.6 Connection negotiation

Upon connecting, the server replies with

1. "CRISP" (as 5 uint32s coding the ASCII values of 'C', 'R', 'I', 'S', 'P');
2. followed by a uint32  $v$  indicating the protocol version number which is comprised of a major version and a minor version. The major version is the upper 8 MSB of  $v$  and the minor protocol version is the 24 LSB of  $v$ .

Some servers may want to protect access by requiring a key. If they do this, then they can simply wait for the first op from the client. If the first op is not `<key>`, then the server can disconnect the client.

Clients connecting to `<key>`-passed servers can send the `<key>` op followed by the key length in terms of `uint32` atoms (the key must be 4-byte aligned). Then the client sends the key. After receiving the key, the server could just disconnect if it doesn't accept the client.

## 2.7 Adding

Adding adds permanent constraints to the solver on the server side. Each constraint is in the form of a clause, which is a disjunction of literals.

When the client adds clauses, it sends the `<add>` op followed by a list of clauses, where each clause is a null-terminated list of literals. When it is done adding clauses, it sends a `<end>`.

The server does not respond to `<add>` or `<end>` ops.

## 2.8 Assuming

When the client makes assumptions, it sends the `<assume>` op followed by a null terminated list of literals. The server does not respond to this op.

The server takes into account the assumptions temporarily, only for the next call to solve. Subsequent calls to solve after the next call are not effected.

## 2.9 Solving

Solving is the only part of the protocol which involves several round trips between the client and the server.

After the client sends `<solve>`, it reads from the server in a blocking read until the server sends a response. The response is either `<sat>`, `<unsat>`, `<unknown>`, or `<end>`.

If the response is `<unknown>`, then the client must respond with either `<continue>` or `<end>`. If the client sends `<continue>`, the protocol enters the same state as if the client had just sent `<solve>`. If the client sends `<end>`, the protocol exits the `<solve>` state; the client can add more clauses or make more assumptions and try to solve again (or quit).

If the server response is `<sat>`,`<unsat>`, or `<end>` then client must not respond with `<continue>` or `<end>`. Instead, the client may optionally request models if the response was `<sat>`, or optionally request failed assumptions if the response was `<unsat>`. In any event, the client may continue to `<add>` or `<assume>` or may `<quit>`.

## 2.10 Models

Models are valuations of variables in a problem which satisfy the permanent constraints and assumptions in the previous `<solve>` interaction. CRISP sup-

ports 2 mechanisms for models: partial and complete models, corresponding respectively to the `<modelfor>` and `<model>` operators.

The `<model>` and `<modelfor>` operators can only be sent if the previous `<solve>` operator ended with a `<sat>` operator from the server and no `<add>` or `<assume>` has taken place since. Otherwise, the server sends `<error>` and disconnects.

Complete models are obtained using the `<model>` operator. The client sends `<model>` and the server responds with a truth value for every variable, in the order of variable index/id 1,2,3,... up until the maximum variable used in any added clause or assumption.

The encoding of the truth values is the same for partial and complete models, and is described below.

Partial models are obtained by using the `<modelfor>` operator. The client sends `<modelfor>` followed by a null terminated list of literals. The server responds with the truth value for each of these literals in the same order specified by the client.

In both `<model>` and `<modelfor>`, the list of truth values is encoded by the server as follows. First, the server sends a `uint32` indicating how many `uint32`s are to be sent subsequently to communicate the model. Then:

Let  $N$  be the length of the list, and let  $M$  be  $\lceil N/32 \rceil$ . Then  $M/32$  `uint32` values are sent by the server. Let  $u_i$  be the  $i$ 'th such value. The truth value for element  $j$  of the list is

$$u_{j/32} \&(1 \ll j \bmod 32) \neq 0$$

Thus models are communicated in a standard compressed bitvector representation with 32 bit word size.

## 2.11 Failed Assumptions

Failed assumptions are a subset of the assumptions from the last `<solve>` sequence which are sufficient to render the problem `unsat`. Like the `<model>` and `<modelfor>` operators, there are 2 failed operators, `<failed>` and `<failedfor>`.

`<failed>` retrieves a sufficient subset of all previously assumed literals to render the last problem posed to `<solve>` `unsat`.

`<failedfor>` specifies a list of assumptions which are of interest to the client. The server responds with the maximal subset of this list which intersects what would be returned by `<failed>`.

Thus, when the client sends `<failedfor>` it follows this operator with a null terminated list of literals. But when the client sends `<failed>` it immediately waits the response from the server.

In both cases, the server replies with a null terminated list of literals. If the server replies to `<failedfor>`, the list of literals in the response from the server must respect the order of the list provided by the client in `<failedfor>`.

## 2.12 Error Conditions

Errors are always sent from the server to the client. Each `<error>` op is followed by a single `uint32` value, which encodes more information about the error.

Anytime the server responds with an `<error>`, the server subsequently simply closes the connection.

The following are common errors:

1. If the `varuint` encoding encodes a number outside the `uint32` range then the server responds with an error. The error code is 1.
2. If the client has this error reading from the server, then it must disconnect.
3. If a client requests a model or partial model and the previous solve did not end with `<sat>`, the server responds with an error. The error code is 2.
4. If a client requests failed assumptions and the previous solve was not `unsat`, the server responds with an error, the error code is 3.
5. If a client requests failed assumptions or model and an `<add>` or an `<assume>` has taken place since the last `<solve>`, the server responds with an error whose code is 4.
6. If a client sends an unknown opcode, the server responds with an error whose code is 5.
7. If the client reads an unknown opcode, it must immediately disconnect.
8. If a protocol opcode is read when data is expected, we have error code 6.
9. Other generic server errors are called "internal server errors", and have error code 7.

## 2.13 Resetting

The `<reset>` opcode causes the server to forget all added clauses and assumptions, *i.e.* to go to the state just after connection negotiation. This is useful in a wire protocol because the overhead of creating a connection is higher than the overhead of creating/initializing an incremental SAT solver via API.

## 3 Extensions

In this section, we mention several possible extensions to CRISP. We have not yet implemented these, but rather present them to show how easy it is to adapt CRISP to different Boolean reasoning and optimisation tasks and solving requirements.

### 3.1 Extension negotiation

The  $\langle \text{ext} \rangle$  opcode indicates a request from the client to find available extensions on the server. Each extension has

1. A unique 16 bit identifier.
2. A list of opcodes.

Thus, in response to  $\langle \text{ext} \rangle$ , the server responds with a null terminated list

$$E = [(i_0, n_0), (i_1, n_1), \dots]$$

where each  $(i, n)$  is placed into a uint32 ( $i$  is the 16 MSB,  $n$  is the 16 LSB). Value  $(i_j, n_j)$  indicates that extension with identifier  $i_j$  has  $n_j$  opcodes.

To map extension opcodes to server instances, the client then computes that opcode  $c$  (zero indexed) in extension at position  $i$  in  $E$  is

$$(2^{32} - 1) - 255 + \sum_{j < i} n_j + c$$

This allows different servers to have different sets of extensions all packed into the 240 extension opcode slots, and it allows clients to map such opcodes<sup>1</sup>.

### 3.2 Optimisation

There are many forms of Boolean optimisation, such as MaxSAT where the problem is to find the maximal number of satisfiable clauses, possibly weighted. Other forms include pseudo-Boolean optimisation, in which constraints and objectives can express the number of literals in clauses which are true. All of these forms can be translated to each other, see for example [3] or [2].

In this extension, we present a tiny, minimalistic incremental Boolean optimisation interface which can also be used to solve a wide variety of Boolean optimisation forms provided the problems are coded accordingly.

This extension defines 5 new opcodes, in the following order.

1.  $\langle \text{addmax} \rangle$ . This opcode adds a literal to the set of literals for which the solver searches for a maximal satisfying assignment. It is used by the client anywhere where  $\langle \text{add} \rangle$  or  $\langle \text{assume} \rangle$  can be used. There is no response from the server.
2.  $\langle \text{clearmax} \rangle$  This opcode clears the set of literals to maximize. It is used by the client anywhere  $\langle \text{add} \rangle$  or  $\langle \text{assume} \rangle$  can be used. There is no response from the server.

---

<sup>1</sup>This mechanism does require that extensions have unique identifiers. We recommend filing an issue with the Gini project on github to request extension identifiers and definitions. Short of that, one could always just choose a random number for the a new extension identifier to minimize the chances of collision.

3. `<nextlower>` This opcode directs the solver to find the next lower bound on the number of maximisation literals which can be satisfied. It has semantics like `<solve>` in that it enters a communication loop between the client and the server. That loop behaves exactly as in `<solve>`, except it replaces `<sat>` with `<better>` (see below).
4. `<nextupper>` This opcode directs the server to find the next upper bound on the number of maximisation literals which can be satisfied. Otherwise, it operates exactly like `<nextlower>`.
5. `<better>`. This opcode replaces `<sat>` in a solver communication sequence for `<lowerbound>` or `<upperbound>`. It is sent from the server when a satisfying assignment increases the lower or upper bound in response to `<nextlower>` or `<nextupper>`. Unlike `<sat>`, It is followed by the respective bound, coded as another uint32. Also unlike `<sat>`, it does not undo the assumptions.

### 3.3 Assumption Multiplexing

One interesting possibility which arises when interacting with a SAT problem is partitioning and multiplexing single hard problem instances across several cores or machines. In this section we present an extension which allows partitioning problems with assumptions and asking the server to multiplex the solving across different sets of assumptions.

#### 3.3.1 Opcode `<mux>`

This extension introduces a new opcode called `<mux>`, which is used by the client and the server to multiplex exchanges on top of the base protocol. In principal, it can also be used for other extensions, though we do not detail that here.

The `<mux>` opcode is a sort of meta code in that it is (almost) always followed by an identifier  $i$  and another opcode  $o$ . The identifier is used to identify which of several multiplexed operations are involved in subsequent exchanges between the client and the server. The client uses `<mux>` only for assumptions and solving. For example, to request multiplexed solving under  $x$  and  $\neg x$ , the client might send

```

<mux> 1 <assume> x 0
<mux> 2 <assume> ¬x 0
<mux> 1 <solve>
...
<mux> 2 <solve>

```

After a send of `<mux> n <solve>`, the client interacts with the server for the solve sequence as usual (*i.e.* without `<mux>`), with the following exceptions.

1. The server prefixes all immediate post-solve replies with `<mux> n` for the appropriate  $n$ , this includes `<sat>`, `<unsat>`, `<unknown>` and `<end>`<sup>2</sup>.

---

<sup>2</sup>In fact, it is not strictly necessary for the server to provide this information because the

2. The client must respond to  $\langle \text{mux} \rangle n \langle \text{sat} \rangle$  with  $\langle \text{model} \rangle$ ,  $\langle \text{modelfor} \rangle$ , or  $\langle \text{end} \rangle$ . If the client responds with  $\langle \text{end} \rangle$ , the server must not reply.
3. The client must respond to  $\langle \text{mux} \rangle n \langle \text{unsat} \rangle$  with  $\langle \text{failed} \rangle$ ,  $\langle \text{failedfor} \rangle$  or  $\langle \text{end} \rangle$ . If the client responds with  $\langle \text{end} \rangle$ , the server must not reply.
4. The client may, in addition to sending  $\langle \text{continue} \rangle$  or  $\langle \text{end} \rangle$  as usual in response to a server supplied  $\langle \text{mux} \rangle n \langle \text{unknown} \rangle$ , either
  - (a) send a new  $\langle \text{mux} \rangle m \langle \text{solve} \rangle$  for some  $m$  which is not currently in a solve interaction but for which there are assumptions defined. This tells the server to continue solve sequence  $n$  and also multiplex a new solve sequence  $m \neq n$ .
  - (b) or, the client may send  $\langle \text{mux} \rangle m \langle \text{assume} \rangle$  for some  $m$  not in a solve sequence. This tells the server to continue solve sequence  $n$  and also to record a new set of assumptions.

In this way, the client may queue parallel solving piecewise to the server, by making a request for 2 solve steps in response to a  $\langle \text{unknown} \rangle$ .

### 3.3.2 Example Trace

To continue the example above, the following is a possible trace between the client and the server exercising the various options above.

N	who	message	queue (mux-id, step)
1	client	$\langle \text{mux} \rangle 1 \langle \text{assume} \rangle x 0$	$\emptyset$
2	client	$\langle \text{mux} \rangle 2 \langle \text{assume} \rangle \neg x 0$	$\emptyset$
3	client	$\langle \text{mux} \rangle 1 \langle \text{solve} \rangle$	$[(1, 1)]$
4	server	$\langle \text{mux} \rangle 1 \langle \text{unknown} \rangle$	$\emptyset$
5	client	$\langle \text{continue} \rangle$	$[(1, 2)]$
6	server	$\langle \text{mux} \rangle 1 \langle \text{unknown} \rangle$	$\emptyset$
7	client	$\langle \text{mux} \rangle 2 \langle \text{solve} \rangle$	$[(2, 1), (1, 3)]$
8	server	$\langle \text{mux} \rangle 2 \langle \text{unknown} \rangle$	$[(1, 3)]$
9	client	$\langle \text{mux} \rangle 3 \langle \text{assume} \rangle y 0$	$[(1, 3), (2, 2)]$
10	server	$\langle \text{mux} \rangle 1 \langle \text{unknown} \rangle$	$[(2, 2)]$
11	client	$\langle \text{mux} \rangle 3 \langle \text{solve} \rangle$	$[(2, 2), (3, 1), (1, 4)]$
12	server	$\langle \text{mux} \rangle 2 \langle \text{sat} \rangle$	$[(3, 1), (1, 4)]$
13	client	$\langle \text{model} \rangle$	$[(3, 1), (1, 4)]$
14	server	$\dots \neg x \dots$	$[(3, 1), (1, 4)]$
15	server	$\langle \text{mux} \rangle 3 \langle \text{unsat} \rangle$	$[(1, 4)]$
16	client	$\langle \text{end} \rangle$	$[(1, 4)]$
17	server	$\langle \text{mux} \rangle 1 \langle \text{unknown} \rangle$	$\emptyset$
18	client	$\langle \text{end} \rangle$	$\emptyset$
19	server	$\langle \text{mux} \rangle 1 \langle \text{end} \rangle$	$\emptyset$

response sequence is a deterministic function of the client request sequence, but providing this information simplifies client implementations and provides the possibility of an additional sanity check

### 3.3.3 Work Queue – Order and Size

With muxing, the server maintains a fine grained work queue which is ultimately completely controlled by the client. There is exactly one interaction in the protocol which allows the client to request more than one response from the server, which is when the client responds to  $\langle \text{mux} \rangle m \langle \text{unknown} \rangle$  with  $\langle \text{mux} \rangle n \langle \text{solve} \rangle$ . Every time this happens, the server appends the next round of work for  $m$  to the queue followed by the first round of work for  $n$ . The queue is a FIFO and whenever the server has an outstanding work queue and is not expecting a response from the client, the server pops the head of the queue and sends the result.

To keep track of when to expect responses from the server, the client simply needs to keep a counter of the size of the work queue, which increments by 1 on the initial send of  $\langle \text{mux} \rangle n \langle \text{solve} \rangle$ , increments by 2 every time the client sends this in response to  $\langle \text{mux} \rangle n \langle \text{unknown} \rangle$ , and decrements by 1 every time the client receives a muxed response from the server. For example, after line 14 in the trace above, the client knows that there are 2 elements in the work queue, so it waits to read from the server before sending any new requests. Similarly, the server knows that whenever the work queue is non-empty and it sent a response for a model or failed literals, the client will not make a request.

### 3.3.4 Mux Queue bounds

By convention, the client may send  $\langle \text{mux} \rangle 0$  to the server to query the maximal size of the work queue. The server responds with a single `uint32`. This number may be useful for partitioning algorithms.

### 3.3.5 Mux Summary

In summary, in terms of the overall flow of the base protocol outline in Section 2.2, muxing preserves a similar flow and corresponding constraints for each multiplexed solution (and extraction) interaction. However, the client *must* respond to  $\langle \text{sat} \rangle$  and  $\langle \text{unsat} \rangle$  even if it does not expect a model. This is necessary so that after each step of the protocol who reads and who writes to the connection is well defined and there is only one reader.

## 3.4 Clause Sharing

A common approach to parallel sat solving is running independent solvers with different configurations or orderings of data structures and then to have them share learned clauses. It would be beneficial to have an extension which enabled this to work in a distributed fashion as well. In fact, this has already been implemented and evaluated with `mpi` (message passing interface). However, `mpi` is meant for scientific cluster computing and each implementation uses different message encodings. A simple agreed upon protocol for clause sharing would enable different solvers to interact in a wider variety of computing environments.

Hence, we propose a crisp extension called clause sharing. The extension uses the `<add>` encoding of clauses to publish and retrieve clauses. We define 4 new protocol points for this extension.

1. `<share>` This protocol point is injected by the client immediately after she is done sending `<add>`. It is a directive to the server to share clauses with the client for the problem defined by the previous `<add>`. The server responds with `<ok>` if it is willing to run a solver or communicate with another solver to share clauses. Otherwise, the server responds with `<end>`.
2. `<putc>` This protocol point is injected by the client during a solve loop, whenever the client may otherwise send `<continue>`. It is followed by a set of clauses encoded as in `<add>` (ie ended with `<end>`). The server simply interprets this as an implicit `<continue>` and may answer as in the core CRISP protocol
3. `<getc>` This protocol point is also injected by the client whenever she may otherwise send `<continue>`. The server responds with a set of clauses encoded as in `add`, ending with `<end>`. The server may also respond with `<sat>`, `<unsat>`, or `<end>` instead of with a set of clauses.

In this case, the client may in fact be a solver. The server may also be

## 4 Experiments

We measured the sizes of varint encoding based compression compared to dimacs format and gzip'd dimacs format.

## 5 Conclusion

We have presented an efficient wire protocol for incremental sat solving which can be used to relegate SAT solving in applications to an external server, possibly with dedicated hardware. The protocol is simple, compact, minimizes round trips, provides compression, and is easily extensible to various domains. From the client point of view, it can be used as a drop in replacement for incremental sat applications.

From a solver developer point of view, the server side is completely isolated from the calling application. As a result, a solver developer can use arbitrary hardware, programming language or style, and make arbitrary use of operating system resources independent of the application.

Moreover, the protocol provides a simple, useful piece of infrastructure for distributed solving, as any solvers which are integrated into a server (or can serve the protocol themselves) can be used to solve sub problems of a hard problem with assumptions. Moreover, the client-server paradigm makes this composable: the server can again be a client to make use of other servers.

We would like to thank Daniel Leberre and Armin Biere for helpful discussion during the development of CRISP.

## References

- [1] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.
- [2] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *JSAT*, 2(1-4):1–26, 2006.
- [3] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.