SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **00**:1–16 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Pedro: A publish/subscribe server using Prolog technology

Peter J. Robinson[1,†] and Keith L. Clark[2]

[1] *School of Information Technology and Electrical Engineering, The University of Queensland*
[2] *Department of Computing, Imperial College*

SUMMARY

Pedro is a TCP/IP publish/subscribe server implemented in C that uses Prolog technology for testing subscriptions against notifications. It provides both content and address based routing of messages between processes. An inter-process message M is a string representation of a Pedro term (usually but not necessarily a variable free term). A subscription is essentially a message template T paired with a Pedro query that is used to restrict the allowed values for variables in T. Pedro automatically forwards any notification it receives to all processes that have a current subscription that covers the notification. A Pedro connected process can optionally register a name with Pedro giving the process a unique Pedro handle. Pedro handles can be used to send peer to peer messages between processes.

In this paper we illustrate the use of Pedro and describe its implementation.

KEY WORDS: publish, subscribe, agent communication, content based message routing, peer to peer communication

1. INTRODUCTION

Asynchronous message transport systems that support publish and subscribe message routing [1] have been found useful for developing open distributed applications [5], [13]. They are particular useful for applications such as enterprise integration [2], complex event processing [11], and our own area of interest, open multi-agent systems [17].

In an open multi-agent system, how does an agent discover other agents that offer services or information it wants to access? In his description of the contract net protocol [16], Smith called this the *connection* problem. For the contract net protocol there is an assumption that all agents are connected to a communication server that supports both broadcast and receiver identified communication. In the case of the contract net, the receiver identity can be be given as a wild card pattern $\star$, indicating that the message should be broadcast to *all* currently connected agents. This is how calls for proposals to

*Correspondence to: Peter Robinson, School of Information Technology and Electrical Engineering, The University of Queensland
†E-mail: pjr@itee.uq.edu.au

Copyright © 2009 John Wiley & Sons, Ltd.

execute a task are usually sent. The proposal includes the registered identifier of the sender, so that a bid response can be sent directly back to the proposer. Of course, there is requirement that all the agents use the same ontology[6] for the content of the messages.

As a simple variant of this, we might allow agents participating in the contract net protocol to subscribe for task announcements of interest to them. The communication server then just forwards a task announcement to those agents who have lodged a subscription that matches the announcement. Such a communications server supports both content routing of messages *and* agent to agent communication using agent identifiers.

In our experience, having both types of message routing infrastructure greatly facilitates the building of distributed multi-agent applications. In [4] and Section 2.2 we describe an English ascending auction simulation in which bidders participate in multiple auctions. Bidders and auctioneers can join the system at any time. Bidders join by lodging subscriptions for bid calls for items of interest to them, up to a maximum call price. Auctioneers join by broadcasting a call for bids for a specific item at a given call price. The bid call includes the auctioneer's identity. This is routed to all bidders with a current subscription that covers the call. A bidder responds by directly sending a bid message to the auctioneer. The bid includes the bidder's identity. The auctioneer acknowledges the first bid message it receives at each call price level by a private acknowledgement sent directly to that bidder.

A publish/subscribe server also provides one of the key functionalities of a KQML matchmaker [10]. This is the handling of KQML [7] messages using the `recruit` performative. These are messages that contain, as their content, another KQML message. This might be an `ask` message requesting an answer to a query, or a `request` message requiring a task to be performed. The matchmaker forwards the embedded message to all agents who have advertised their interest in handling messages that match the content of the `recruit`. The KQML `advertisement` message, whose content is a message pattern, is essentially a subscription for `recruit` messages. When using a publish/subscribe server, every message sent to the server is treated as though it were wrapped in a `recruit` message.

We have for several years been using Qu-Prolog, which is a multi-threaded Prolog, for implementing multi-agent applications [4]. Until recently we had used Elvin [14] publish/subscribe servers for content based message routing, and McCabe's ICM servers [12] for addressed agent-to-agent communication.

Elvin notifications are sets of attribute/value pairs and subscriptions are boolean tests on attribute values using the Elvin repertoire of primitive operations and predicates. It is no longer available for non-commercial use. Because our primary use of Elvin was in Prolog applications, and we have expertise in Prolog implementations, we decided to implement our own publish/subscribe server in C but with a Prolog technology component. This allows us to do contents routing based on *unification* of a notification against a notification template and successful evaluation of a query. The query tests values of variables in the message template generated via the unification against a notification.

Pedro messages are strings representing Pedro terms. For example, a subscription is a string S of the form

```
subscribe(T,Q,R)
```

Here T is a notification template (a Pedro term, usually with variables), Q is an associated Pedro query, using variables that appear in T, and R is an integer. We call R the rock for the subscription. Pedro automatically forwards any notification message it receives to each process P that has a current

Copyright © 2009 John Wiley & Sons, Ltd.

subscription `S` that *covers* the notification. It will *cover* a notification `N` iff the Pedro query `T=N,Q` succeeds. Here, = is unification[15].

The forwarded notification is preceded by the rock value contained in `P`'s subscription. This can then be used by `P` to switch the notification to an appropriate message buffer or internal thread. If there is no subscription that covers a notification, it is discarded by Pedro.

Soon after we implemented the first version of Pedro, the ICM messaging infrastructure became an unsupported system. ICM provided us with peer-to-peer communication with receiver addresses similar to email addresses – using the machine name, process name and, optionally, a thread name within the process, to identify the destination for a message. Since peer to peer messaging using registered agent identifiers is widely used for agent communication - it is assumed in the KQML [7] and FIPA ACL [8] agent communication languages - we wanted Pedro to additionally support this form of communication.

To participate in publish/subscribe communication a process on a host `H` must connect with some Pedro server. In addition, the process can optionally register a name `P` with the Pedro server. Only one process per host can register a given name. Registration of the name is tantamount to `P` lodging a special subscription:

```
subscribe(p2pmsg(P@H,_,_),true,...)
```

for `p2pmsg` notification terms which have `P@H` as first argument. We call `P@H` the Pedro handle of the process. Note that because Pedro allows only one process per host to use the name `P` this subscription is unique to the `P` process on `H`. However, a process on a different host can register the name `P`.

Suppose another process on a host `H1` connects with the same Pedro server and registers the name `P1`. This process can send a message `M` just to `P` on `H` by sending a notification of the form `p2pmsg(P@H,P1@H1,M)` to Pedro. The second argument of the `p2pmsg` term is the Pedro handle of the sender so that the receiver can send a `p2pmsg` back to the sender, if need be.

This way of implementing peer to peer communication actually generalizes that provided by the ICM. A notification of the form

```
p2pmsg(_@texel03,...,...)
```

will be forwarded to all processes that have registered any name with Pedro on host `texel03`. A notification of the form:

```
p2pmsg(worker@_,...,...)
```

will be forwarded to all processes that have registered the name `worker` with Pedro on different hosts. Finally, a notification of the form

```
p2pmsg(_,...,...)
```

will be forwarded to any process, on any host, that has registered a name with the Pedro server to which the notification is sent. In this case the `_` has the role of the `*` of the contract net protocol. Note that posting a notification `p2pmsg(_,P@H,M)` is a way of routing a message `M` to all name registered processes even if they have not lodged a subscription that covers `M`. The processes that have not registered a name will not receive the notification.

Our overall objective in implementing Pedro was to:

Copyright © 2009 John Wiley & Sons, Ltd.

- provide a lightweight easily ported communication server with a simple API; and
- use Prolog implementation technology for efficient but high level content based routing of structured information.

We currently have API's for Qu-Prolog, C, Java, Python and TCL/TK.

In the next section, we illustrate the use of Pedro for quickly building hybrid distributed applications. We then more formally describe the syntax of notifications and subscriptions and Pedro notification routing semantics. After that we discus its implementation, especially as it relates to the use of Prolog technology. Next we compare Pedro with other communication systems and Prolog. The paper concludes with a brief discussion of future work. We assume some familiarity with Prolog and its abstract machine implementation.

2. PEDRO APPLICATIONS

We are currently exploring the use of Pedro in multi-agent, event processing and cognitive robot applications.

2.1. Event processing and distributed control

The event processing application involves the simulation of a house in which there are sensors and control devices. The sensors issue readings periodically, or in response to events, such a person entering a room. The events are primarily caused by interacting with the house simulation using a GUI. Via this interface we can explore various activity scenarios. The GUI and the house simulator are implemented in Python, which watches for mouse events and maps these into house state updates.

The simulator periodically sends sensor readings as notifications to Pedro. These are routed to control agents that have subscribed for notifications of interest to them. These agents are external to the house simulation, and are implemented in Qu-Prolog. The agents periodically post control messages as notifications to Pedro. These will be routed back to the simulator because the sensor and control device objects inside the Python house simulator subscribe for the control messages they can handle. The agent sent control messages also update the state of the simulated house.

As an example, the temperature in a room such as the `kitchen` might normally be sent every 30 minutes as a `temp` notification. The one sent at 10.30 might be

```
temp(kitchen,25,10:30)
```

The above notification is routed to the kitchen agent, that has lodged the subscription

```
subscribe(temp(kitchen,Temp,_), (Temp>20),...)
```

Because the temperature is a little hot, this agent will respond by issuing a notification

```
change_notification_frequency(temp(kitchen,_,_),5))
```

This is its request to the kitchen temperature sensor to send out temperature readings every 5 minutes until further notice. If the sensor is configurable in this way, and say, is capable of sending out temperature readings at a frequency between every minute and every thirty minutes, the sensor will have lodged a subscription

Copyright © 2009 John Wiley & Sons, Ltd.

```
subscribe(change_notification_frequency(temp(kitchen,_,_),F)),
                  (1=<F,F=<30),...)
```

If so, it will receive the notification asking for a change of frequency in its `temp` notifications. If not, the frequency change request is discarded by Pedro and the temperature notifications will continue at 30 minute intervals.

Now, suppose the kitchen agent receives two consecutive kitchen temperature readings at 25 or above, at the times 10.35 and 10.40. The kitchen monitoring agent will react by issuing a notification

```
cool(kitchen)
```

and an inferred event notification

```
possible_failure(radiator_temp_control,kitchen)
```

The `cool(kitchen)` notification will perhaps be picked up by a window opening device in the kitchen, and the `possible_failure` message by a fault monitor agent that will log the event.

After several temperature notifications have fallen below 25 degrees, the kitchen agent can then post the notifications

```
change_notification_frequency(temp(kitchen,_,_),15)
unrequest(cool(kitchen))
```

The key advantage of our using Pedro for an application which involves both event processing and control responses, is that all that has to be decided is the ontology for event notifications and control messages. The system is then open. As sensors or control devices are added they subscribe to Pedro for the control messages they can handle. As monitoring agents are added they subscribe for the event notifications of interest to them, which they can update in response to events. The agents then attempt to exert control over the monitored system by issuing control notifications. No component - agent, control device or sensor - needs to know the identities of other components, or even what other components there are.

2.2. Bidding agents in multiple auctions

This application is implemented in Qu-Prolog with a visualization in TCL/TK. It uses both the publish/subscribe and peer-to-peer messaging provided by Pedro. The application was originally implemented using Elvin and ICM communication servers and this earlier implementation is described in [4].

Each bidding agent has a set of items it would like to purchase in each auction, which is conducted as an English ascending bid auction. All the agents, auctioneers and bidders, register a name with Pedro.

Auctions can start at any time and bidders can join an auction at any time. A bidder joins an auction by lodging a subscription for `bid_call` notifications for each item they want to purchase in that auction with a upper limit on the call price. The subscription is their 'ear' for bid calls for these items. So, if a bidder is interested in an item with lot number 123, but is not prepared to pay more than 400 pounds for it, it sends a subscription to Pedro such as

```
subscribe(bid_call(lot(123),price(P),_),(P=<400),..)
```

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using speauth.cls

If it has arrived after the item has been auctioned, or after the bidding has risen above 400, the bidder will not receive any bid_calls for the item. Likewise, if it is participating in the auction for the item, and the bid call price rises above 400, it will not receive any more calls from that auctioneer until it starts auctioning another item the bidder wants to buy.

An auction starts when its auctioneer posts a bid_call for a specific item at an initial call price. If a bidder receives such a bid_call notification, say

```
bid_call(lot(123),price(350),auctioneer(fred@pictor))
```

it has to decide quickly whether or not to bid. Since it is competing in multiple auctions it might not have sufficient uncommitted funds to bid at this time, and so might have to skip a round of bidding for this item. If it is able to bid, it sends its bid response directly to the auctioneer whose Pedro handle has been given in the bid_call notification using a p2pmsg notification. This will include the bidder's Pedro handle, say bill@zeus. So, a bidder bill@zeus would send the notification

```
p2pmsg(fred@pictor,bill@zeus,bid(lot(123),price(350)))
```

to bid for lot 123 at call price 350. If this is the first bid to be received at this price, the auctioneer replies with the p2pmsg:

```
p2pmsg(bill@zeus, fred@pictor, ack_bid(lot(123),price(350)))
```

This tells bill@zeus that it need not bid in the next round, and that it will win the auction if there are no bids in response to the next bid_call. The auctioneer ignores and discards all bid messages it receives at a given call price, except the first.

The above application makes use of a typical interaction protocol. The auctioneer broadcasts a call for bids including in the call its Pedro handle which can be used for sending a peer-to-peer message response. The call is routed to all interested agents, who then respond by a message routed just to the auctioneer. This is the protocol of the contract net. As we mentioned in the introduction, the routing of a broadcast message using subscriptions to agents interested in its content is a way that agents can connect. After the connection is established, they can communicate directly.

2.3. Distributed querying

Imagine a network of agents each of which holds partial information about some set of relations and is prepared to be queried and updated about such relations. The agents then exchange information using ask and tell messages. An ask may be sent as a peer-to-peer message if the querying agent knows the identity of an agent that can provide an answer, via a tell. The ask may also be sent to Pedro as a notification which is routed to those agents that have implicitly 'advertised' they can respond to the ask by lodging a covering subscription.

As a very simple example, assume different agents hold floor area information about different rooms in a building expressed as facts and rules about an area(Room,SqFeet) relation. The agents kitchenAg@corsair and bedroomAg@texel, that respectively are willing to be asked about the the areas of the rooms kitchen and bedroom, can lodge subscriptions

```
subscribe(ask(area(kitchen,_),_),true,..)
subscribe(ask(area(bedroom,_),_),true,..)
```

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

Another agent, `heatingAg@russet`, wanting to collect information about room areas, can send the following notification to Pedro

```
ask(area(R,A),heatingAg@russet)
```

with its Pedro handle included. This will be routed, as sent, to both `kitchenAg` and `bedroomAg` as their subscriptions each have a notification template that unifies with the notification. As an example response, the `kitchenAg` might reply with a `p2pmsg` notification such as

```
p2pmsg(heatingAg@russet,kitchenAg@corsair,tell(area(kitchen,1200))
```

to be routed directly to the querying `heatingAg@russet` agent.

As an extension of this information exchange by `ask` queries and `tell` responses, let us suppose that we allow agents to periodically announce new information when they acquire it, particularly when this can be expressed as a general rule of the form

```
rule(Conclusion,Conditions)
```

An agent interested in any rule about the `area/2` relation can lodge a subscription

```
subscribe(tell(rule(area(_,_),_),true,..)
```

Now, when another agent acquires a new rule about `area/2` it can disseminate it to all agents who are interested in receiving such a rule, using a notification

```
tell(rule( area(R,A), (rect(R),length(R,L),width(R,W),A is L*W)) )
```

Note that in this example application notifications as well as the subscriptions contain variables.

3. PEDRO MESSAGE SYNTAX AND ROUTING SEMANTICS

The Pedro server typically runs as a daemon with either the default or a supplied port number used for client connections. Clients connect using this port number and, using the connection protocol, the client and server create two sockets for communication – one for data and one for acknowledgements. The client then uses the data socket for sending subscriptions and notifications. The server responds to such messages by sending an acknowledgment. The server also uses the data socket to send notifications to clients with covering subscriptions. In principle, we could have used a single socket for both acknowledgements and notifications. This would have required us to use a special string representation of acknowledgements to ensure they could not be confused with application level notifications. Furthermore, clients would have to search for acknowledgements within the notifications stream.

All (valid) messages sent between clients and the server are newline terminated strings representing Pedro terms, a restricted set of Prolog terms. The message syntax allows the use of a fixed collection of standard operators such as arithmetic operators, `is`, conjunction, disjunction and if-then-else as well as the infix operators `:` and `@` that are used to construct Pedro handles. This means that it is simple to write an efficient parser (and writer) for messages because, unlike Prolog, in a Pedro term there are no

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using speauth.cls

user-declared operators to deal with. The server has a limit on the maximum length of a message and will reject messages that are too long or do not parse.

An alternative would have been to use no operators and use just Prolog canonical form terms. However, allowing, for example arithmetic operators, means we have more human readable subscriptions, and the overhead for parsing is low because a fixed set of operators is used. Note that clients can use subscriptions and notifications in canonical form if they choose.

Prolog uses a fullstop followed by a whitespace character to signal the end of a string representation of a Prolog term, while we use a single whitespace character (a newline). In either case care must be taken when the terminator occurs within quoted atoms or strings. For us, clients should use `\n` for such occurrences of newlines.

In Qu-Prolog a fast Pedro term parser and writer are used to process Pedro messages rather than the standard term reader and writer for efficiency.

3.1. Notifications

A valid notification is a string that represents an atom or a compound Pedro term. The Pedro server reads characters from a client until it gets a newline. The server will then attempt to parse the characters up to the newline as a term. If this succeeds then the server processes the notification term and sends a 1 to the client as an acknowledgement; otherwise the string is ignored and a 0 is sent to the client as an acknowledgement. Pedro then forwards the notification to the owner of each covering subscription preceded by the rock given with the subscription.

3.2. Subscriptions

Subscriptions are strings of the form `subscribe(`*`Term`*`,`*`Cond`*`, `*`Rock`*`)` where *`Term`* is a Pedro term, *`Cond`* is a Pedro query and *`Rock`* is an integer. The *`Rock`* value is used by the client for its own purposes. In Qu-Prolog it is the identifier of the subscribing thread. It might also be the identifier of a message buffer, or it might not be needed at all by the client. In which case every subscription can just have a default *`Rock`* value of `0`. When the subscription covers a notification, the supplied *`Rock`* value is attached to the notification when it is sent to the client.

If the subscription is valid – i.e. it parses as a Pedro term and the condition is a valid Pedro query as described below, then the server adds the subscription to its collection of subscriptions and sends a Pedro generated subscription identifier *`SID`* (a positive integer) to the client as an acknowledgement. This uniquely identifies that subscription for that client. Otherwise, a 0 is returned by Pedro, indicating that the `subscribe` message has not been accepted.

If a client later sends a string of the form `unsubscribe(`*`SID`*`)` to the Pedro server, the server will delete the subscription with the identifier *`SID`*, provided it was that client that lodged the subscription. The server will acknowledge with a 1 or 0 depending on the success or failure of the request.

A Pedro query has the structure of a Prolog query but differs in that only a fixed set of primitive predicates may be used and some of these have a subtly different semantics from the corresponding ones in Prolog in order to prevent non-terminating query evaluations. For efficiency reasons, Pedro includes a string type. Strings are stored internally as byte arrays rather than as lists of ASCII values, as in Prolog.

Overall, a Pedro query has the form

- `G1,G2` - conjunction
- `G1;G2` - disjunction
- `G1 -> G2 ; G3` - conditional
- `not(G)` - negation (meaning `G` fails)
- `once(G)` - only find one solution of `G`

where `G`, `G1`, `G2` and `G3` are Pedro queries.

The single condition queries are:

- `true  fail`
- `T1 = T2` - unify `T1` with `T2`
- `T is Exp` - unify `T` with the value of expression `Exp`
- `Exp1<Exp2  Exp1=<Exp2  Exp1>Exp2  Exp1>=Exp2`
- `atom(X)  number(X)  list(X)`
- `member(X,L)` - list membership test, can be used to check or generate bindings for X for a given L
- `string(X)` - succeeds if and only if `X` is a string
- `split(L1, L2, L3)` - `L1` is the concatenation of the lists `L2` and `L3`, can be used to find all splittings of `L1`, or to strip off a given front or back sublist of `L1`
- `splitstring(S1, S2, S3)` - `S1` is the concatenation of the strings `S2` and `S3`, with same uses as `split`

In the above, `Exp` indicates an arithmetic expression. The allowed arithmetic expressions are reasonably standard and are listed in the Pedro manual.

3.3. Covering Test

When a notification is sent from a client, the Pedro server tests each of its current subscriptions to see if it covers the notification.

Suppose `N` was received as the string `NS<newline>`. For each covering subscription with rock `R`, Pedro sends the string `R<space>NS<newline>` to the client that lodged the subscription.

Below are more examples of subscriptions. In these examples, the rock is 0 but could have been, for example, a thread ID.

- `subscribe(_, true, 0)` - covers any notification except `p2pmsg` ones.
- `subscribe(foo(X, X), (number(X), (X < 10; X > 20)), 0)`
  - covers any notification `foo(Y,Y)` where `Y` is a number less than 10 or greater than 20.
- ```
  subscribe(bar(L), (list(L), member(X,[apples,pears,oranges]),
                                member(X,L)), 0)
  ```
  - covers any notification `bar(L1)` where `L1` is a list that contains one or more of the atoms `apples,pears,oranges`
- ```
  subscribe(str(S), (splitstring(S, _, S2),
                     splitstring(S2, "hello", _)), 0)
  ```
  - covers any notification `str(S1)` where `S1` is a string containing `"hello"` in any position

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

An important difference between Pedro and standard Prolog query evaluation is that Pedro does simple type tests before evaluating an arithmetic primitive. For example, Pedro checks if an arithmetic expression contains a variable or a non-numeric value before evaluating it, and *fails* the query if it does. In a normal Prolog evaluation, an unbound variable or non-numeric value in an arithmetic expression being evaluated typically results in an exception being thrown.

The following example illustrates this. Consider the subscription

```
subscribe(foo(X, Y), (X < 0 -> Y > 10 ; Y < 10), 0)
```

and the notification

```
foo(bar, 0)
```

In this case the template of the subscription unifies with the notification but `X` is not a number and so the test `X < 0` produces a type error which causes the attempted cover test

```
foo(bar, 0)=foo(X, Y), (X < 0 -> Y > 10 ; Y < 10)
```

to fail.

If the intention of the subscription was to test if `X` is a number, and if so test if its value is less than 0, then it should be written as follows.

```
subscribe(foo(X, Y), (number(X), X < 0 -> Y > 10 ; Y < 10), 0)
```

In this case the notification `foo(bar,0)` is covered.

In the current implementation, if a client lodges two subscriptions, that both cover a notification, then that notification will be sent to the client twice. This issue is further discussed in Section 6.

3.4. Name registration and peer-to-peer communication

As we have already mentioned, Pedro provides a mechanism for peer-to-peer communication within the subscription/notification framework. This is done by clients registering names with the server. The registered name becomes the name of the client process and can be used by other clients to send peer-to-peer messages.

A message sent to Pedro of the form `register(P)` from a client process on a machine *H* registers *P* as the name for the client, provided there is no current registration from the same host using that name. The client process now has the Pedro handle *P@H*.

Semantically, such a registration message can be thought of as a restricted subscription of the form

```
subscribe(p2pmsg(P@H,_,_), true, 0)
```

The restriction being that there can be only *one* such subscription using *P@H*.

When a registered client with Pedro handle *P1@H1* wants to send a message *M* just to another registered client *P@H* it sends a notification `p2pmsg(P@H,P1@H1,M)` to Pedro. The first argument is the address of the process being sent the message. The second argument is the address of the sender and the third argument is the actual message. When Pedro receives a `p2pmsg` notification, it first confirms that the given sender address *P1@H1* is the address of the notifying process. If it is, it forwards the `p2pmsg` notification to the client process with Pedro handle *P@H*, else it sends a failure acknowledgment to the notifying process, and discards the notification.

In fact, the Pedro handles of both the destination and sending processes can be a little more complex. They can have the form *R*:*P@H*, where *R* is a thread or message buffer identifier within the process, similar to the rock that is given in a `subscribe` message. Pedro ignores this when determining to

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

which processes to forward the message, but the recipient receives the complete `p2pmsg` string and can use the given *R* for internal message routing purposes.

A client can deregister its name *P* by sending a `deregister` notification of the form `deregister(P)` to Pedro.

3.5. Arrival Order

If a client sends two consecutive notifications that both get routed to the same process, Pedro will forward the notifications in the time order they were sent. This is not guaranteed for notifications sent from different clients.

4. PEDRO IMPLEMENTATION

This section discusses the implementation of the Pedro server and in particular the use of Prolog technology. The reader is referred to [9] for a tutorial on the Warren Abstract Machine (WAM). Because Pedro contains neither library code nor user-supplied code, the Pedro abstract machine is much simpler than a Prolog abstract machine. For example, Pedro does not require either X or Y registers or an environment.

The Pedro server imposes a limit on the length of strings sent by clients. The default is 1K bytes but this can be changed using a switch when starting the server. Having a known maximum size for client strings that represent Pedro terms means that the sizes of the Pedro heap, trail and choice point stacks can be set so notifications and subscriptions can be parsed and tested without risk of overflowing a stack or filling the heap. This, coupled with restrictions Pedro imposes on the use of the primitive predicates of subscription tests, means that it does not need a heap garbage collector.

The topics covered in this section are the parsing and representation of Pedro terms, the Pedro abstract machine, notification cover testing and some non-Prolog related issues.

Pedro is single threaded and has a very simple architecture as depicted in Figure 1.

4.1. Pedro Term Representation

Pedro follows Prolog fairly closely in the way it represents terms. Structures, lists, numbers and variables are represented in a standard Prolog way on the heap using tagged data containing values or pointers to terms. A list is simply a “dotted pair” consisting of a head term and a tail term. Strings are represented as a tagged word that includes the string length followed by enough words to contain the bytes of the string.

In Pedro atoms have two different representations. Because of the long-lived nature of the Pedro server it is crucial that atoms be garbage collected. For efficiency reasons atoms are divided into two categories: “dynamic” and “static”. The static atoms are the ones that are expected to have a reasonably long life - i.e. the atoms representing the standard operators, the standard predicate names that can be used in subscription queries, and atoms appearing in subscriptions. These atoms are stored in a standard atom table and are garbage collected using reference counting. So, for example, an atom that only appears in one subscription will be removed when that subscription is removed.

Copyright © 2009 John Wiley & Sons, Ltd.

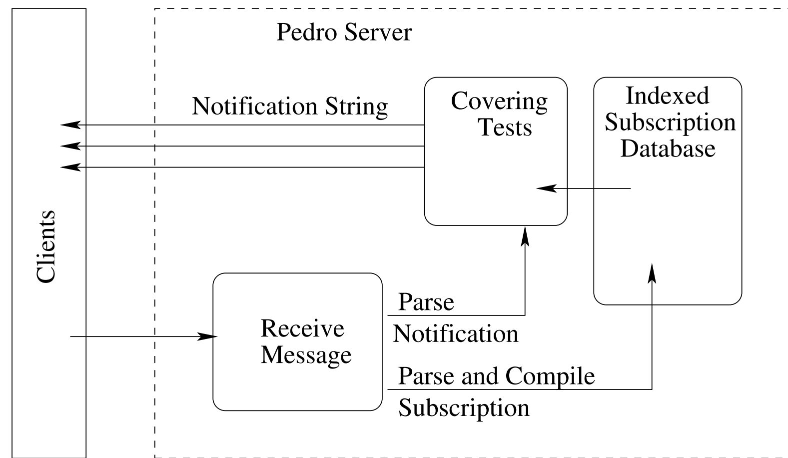

Figure 1. Pedro Architecture

Atoms that appear in a notification which are not in the atom table are only required for cover testing this notification against the subscriptions. These are the dynamic atoms. They are represented in the same way as strings, on the heap. These atoms are automatically removed when the heap is popped on completion of the processing of each notification.

The parser determines which category a given atom belongs to and gives the two types of atoms different tags. This means that atoms with different tags fail to unify. Note that, because the dynamic atoms do not use the atom table, repeated occurrences of the same dynamic atom are not structure shared. The trade-off is that, although more space is required for repeated dynamic atoms, garbage collection of dynamic atoms is more efficient. It was felt that repeated dynamic atoms in a single notification would be fairly infrequent and so the efficiency of garbage collection would be more important than the space efficiency provided by the atom table.

4.2. The Pedro Abstract Machine

Each subscription, just like any other string sent to the Pedro server, is parsed and stored as a term on the heap. The subscription is then compiled to abstract machine code and optimized. Space is then allocated (using the C `alloc`) to store a copy of the parsed term and its abstract machine code. This space is de-allocated when the subscription is removed by the owner client. The allowed predicates for subscription conditions have been chosen so that they can all be implemented directly as a series of machine instructions that cannot lead to infinite computation. The first part of this implies that no predicate definitions are needed so there is no term copying and therefore X and Y registers, environments, put, get, set and unify instructions are not required. The second part is a requirement to avoid one form of denial of service attack. This means, in particular, that occurs checking is needed

Copyright © 2009 John Wiley & Sons, Ltd.

during unification and that list processing tests must fail when trying to access the head or tail of a variable. This is further discussed in Section 5.4.

The abstract machine, in common with a typical Prolog abstract machine, consists of a program counter (PC), a variable binding trail, a choice point stack and a cut point (a pointer into the choice point stack).

Some of the Pedro abstract machine instructions have the same semantics as the equivalent ones in a typical Prolog abstract machine and are listed below.

- `FAIL` - backtrack
- `JUMP` - set the PC to a new location
- `TRY` - create a new choice point and set the next choice
- `RETRY` - reset the next choice
- `TRUST` - remove the choice point created by the try instruction
- `CUT` - reset the top of the choice point stack to the cut point

The other instructions together with their semantics are listed below.

- `EXIT` - backtrack to the initial choice point and exit with success
- `NOOP` - should be unreachable (used by the optimizer)
- `UNIFY` - unify its two arguments (with occurs check)
- `IS` - implementation of `is/2` on its two arguments
- `LT` - ditto for `</2`
- `LE` - ditto for `=<\2`
- `GT` - ditto for `>/2`
- `GE` - ditto for `>=\2`
- `NUMBER` - test if its argument is a number
- `ATOM` - ditto for atom
- `STRING` - ditto for string
- `LIST` - test if its argument is the empty list or a “dotted pair”
- `NOTTRY` - create a choice point for a `not/1` condition
- `IFTRY` - create a choice point for an if-then-else condition
- `MEMBERTRY` - see below
- `MEMBER` - see below
- `SPLITTRY` - see below
- `SPLIT` - see below
- `SPLITSTRINGTRY` - see below
- `SPLITSTRING` - see below

The arguments of the instructions are typically pointers into the subscription term. The `EXIT` instruction is called when the subscription body succeeds and backtracks in order to reset the variables in the subscription term and clean up the trail and choice point stack.

Figure 2 gives examples of the compilation for `member/2`, `split/3` conditions.

The instruction `MEMBERTRY` checks to see if its argument is either the empty list or a dotted-pair and if not backtracks. Otherwise it creates a choice point and copies its argument (the list pointer) into the second argument of the following `MEMBER` instruction.

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using speauth.cls

```
member(X, XS)              split(L1, L2, L3)

MEMBERTRY                  SPLITTRY
pointer to XS              pointer to L1
MEMBER                     SPLIT
pointer to X               pointer to L2
0                          pointer to L3
                           0
                           0
                           0
```

Figure 2. Compilation examples

```
if not dottedPair(arg2)
  pop choice stack
  backtrack
else
  head := head(arg2)
  arg2 := tail(arg2)
  if not unify(arg1, head)
      backtrack
```

Figure 3. Pseudocode for MEMBER

Figure 3 gives pseudocode for the `MEMBER` instruction. The key point is that each time the instruction is called (either for the first time or upon backtracking) the list in the second argument is replaced by the tail of the list. This means that backtracking will cause a next element of the list to be considered for unification with the first argument. Note that when the list becomes empty or a non-dotted-pair, there are no more choices and so the choice point is removed. This gives different semantics from `member/2` in Prolog. The Pedro version fails when presented with a variable instead of a list, whereas Prolog instantiates the variable to a dotted-pair of variables. The Pedro semantics is necessary to avoid a potential infinite computation.

The `SPLITTRY` and `SPLIT` instructions deal with backtracking in a similar way to the above instructions but are more complicated. In this case the `SPLIT` instruction has five arguments. The

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

```
// arg is the argument of SPLITTRY and arg1,arg2,arg3,arg4,arg5
// are the arguments of the following SPLIT instruction

if arg = []
  if unify(arg1, []) and unify(arg2, [])
      pc = pc + 6      // skip over following SPLIT instruction
  else
      BACKTRACK
else if not dottedPair(arg)
  BACKTRACK
else
  arg3 = []
  arg4 = ptr to arg3
  arg5 = arg
  push a choice point
  if unify(arg1, arg3) and unify(arg2,arg5)
      pc = pc + 6
  else
      BACKTRACK
```

Figure 4. Pseudocode for SPLITTRY

first two are the last two arguments of the `split/3` condition. The third argument contains the first part of the list being split and the fifth argument contains the remainder of the list being split. The forth argument contains the tail pointer of the third argument so that the next element taken off the fifth argument can be efficiently added to the end of the third argument.

Figure 4 gives pseudocode for the `SPLITTRY` instruction. If the list being split is the empty list then there is only one possible solution and so no choice point is constructed. Otherwise a choice point is created and the arguments of the following `SPLIT` instruction are initialized ready for backtracking. The first possible solution (splitting the list into the empty list and the entire list) is then tried.

Figure 5 gives pseudocode for the `SPLIT` instruction. This instruction is responsible for generating the next split and attempting the required unification. Note that the top of heap stored in the choice point needs to be updated because of the new dotted pair item created on the heap.

The code for `SPLITSTRINGTRY` and `SPLITSTRING` is similar to the code above but a little simpler with straightforward string processing used instead of the list processing above.

Because Pedro does not require X and Y registers or environments, compilation to the abstract machine is straightforward. Code optimization consists of replacing (chains of) `JUMP` instructions by

Copyright © 2009 John Wiley & Sons, Ltd.

```
if arg5 = []    // no more solutions
  pop choice point
  BACKTRACK
else
  newlist = newDottedPair(heap, head(arg5), [])
  *arg4 = newlist       // add the head of arg5 to the end of arg3
  arg4 = tailptr(newlist)
  arg5 = tail(arg5)
  reset top of heap in choice point to current top of heap
  if not (unify(arg1, arg3) and unify(arg2,arg5))
      BACKTRACK
```

Figure 5. Pseudocode for SPLIT

the code at the destination provided that code takes up no more space than that taken by the `JUMP` instruction. This typically turns out to be an `EXIT` instruction.

4.3. Non-Prolog Issues

The Pedro server has been designed to protect against denial of service attacks that are either accidental or malicious. As described above, restricting the size of client strings, using occurs checks during unification and not allowing subscription conditions that could lead to infinite computation go some way to preventing such attacks.

The Pedro server also uses timeouts to prevent two other forms of attack. If a client does not complete the connection protocol within a specified time then the client is disconnected.

If a client does not consume notifications sent to it then the TCP/IP buffer for that client will fill up, blocking notifications being sent to all clients and eventually blocking other clients from sending notifications. If the client continues not to consume notifications then the server disconnects the client within a specified time.

The Pedro system comes with C, Python, TCL/TK and Java APIs to provide support for client writers. This is particularly useful for multi-agent applications where a visualization or a non-Prolog model of the world is required. Use of Pedro is supported by special syntax in Qu-Prolog.

In terms of performance, we carried out a series of simple tests on a dual core 2.4GHz Intel processor host running Pedro and two clients. The first test exchanged peer-to-peer messages between the two clients at the rate of over 20000 messages per second. This was the same for both Prolog and C clients. The message rate was also the same when the messages were routed using a single covering subscription, requiring just a simple unification test. When 1001 subscriptions needed to be tested for

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

each notification, with only one succeeding, where each failing test involved both a simple succeeding unification and a failing `member/2` test against a list of length three, the message transfer rate was reduced to a little over 2600 messages per second. For most applications one would not normally have to test each notification against 1000 subscriptions. However, it does show that as the number of subscriptions increases, Pedro processing time becomes significant compared with the TCP/IP communication time.

In the early stages of Pedro development we did some simple performance comparisons with Elvin and obtained similar results for message exchange. We were not able to compare the most recent version of Pedro with Elvin as we no longer have access to Elvin.

5. COMPARISONS

In this section Pedro is compared with Elvin, ICM, and xmlBlaster, which takes a rather different view on what notifications and subscriptions are. We also discuss the use of Prolog for an alternative implementation for Pedro.

5.1. Detailed Comparison with Elvin

Elvin is a sophisticated subscription/notification system designed for use on a wide-area network. It supports a federation of Elvin servers – with servers not only sending notifications to clients but also to other servers for further distribution. Clients can use server discovery to determine locations of Elvin servers. For agent applications on a local-area network these features were not considered as important and so were not implemented in this version of Pedro.

Elvin clients need to use the API to encode and decode communications with the Elvin server – these communications are not human-readable. By contrast, all communications with the Pedro server are human-readable strings.

Elvin notifications are sets of attribute/value pairs. Values are numbers, strings or opaque, with strings the only structured values that can be examined by a subscription test. Elvin subscriptions are boolean valued expressions involving attribute names and a set of predefined predicates for testing values.

As an example, we consider a slight variant of the `bid_call` notification and subscription of Section 2.2 which now includes, in the subscription, a pattern identifying the auctioneer. The Pedro notification

```
bid_call(lot(123),price(350),auctioneer(fred@pictor))
```

can be represented as the following Elvin notification.

```
{msg_name/"bid_call", lot/123, price/350, auctioneer/"fred@pictor"}
```

The Pedro subscription

```
subscribe(bid_call(lot(123),price(P),auctioneer(fred@_)),(P=<400),..)
```

can be represented as the following Elvin subscription.

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

```
msg_name == "bid_call" && lot == 123 &&
begins_with(auctioneer, "fred@") && price <= 400
```

Note that the Elvin subscription is a logical expression with the auctioneer name checked by string processing. The Pedro equivalent is a pattern and a logical expression, where the auctioneer name is checked by unification. To avoid this string processing in Elvin, the auctioneer information could be flattened to two attribute/value pairs.

One advantage of Elvin is that the attribute/value pairs in a notification can be in any order. Also, additional attribute/value pairs in the notification with the attribute not mentioned in a subscription may still pass the subscription test. This sort of flexibility can be achieved in Pedro by using lists of attribute/value pairs and by using member tests.

For example, the above Pedro `bid_call` term could have been sent as the list of attribute/value pairs

```
[msg_name/bid_call, lot/123, price/350, auctioneer/fred@pictor]
```

and the above Pedro subscription regarding such notifications is

```
subscribe(L, (member(msg_name/bid_call, L), member(lot/123, L),
              member(price/P, L),  member(auctioneer/fred@_, L),
              P =< 400)), ..)
```

This is more verbose than the subscription for `bid_call` terms, but has the same structure as the Elvin equivalent and the subscription will now cover notification lists with any order of the attribute/value pairs. It will also cover a notification containing extra pairs such as a `bid_by` pair that gives a time by which bids must be submitted.

For our distributed applications using Qu-Prolog, the nested term representation rather than the list representation is generally used because it allows a client to use unification to branch on and extract values from received notifications, rather than list membership tests.

A major difference between Pedro and Elvin is that Pedro notifications can contain variables which may be instantiated during the unification of a covering test, even though the notification is forwarded uninstantiated. This is illustrated by the `ask/tell` example of Section 2.3.

The `ask` notification `ask(area(R,A),heatingAg@russet)` is covered by both the `kitchenAg` and `bedroomAg` subscriptions with the variable `R` instantiated during the unification part of the test. One way to achieve this in Elvin is to allow fields to have an application defined special unknown value such as `"?"`. The corresponding Elvin notification could then be given as

```
{msg_name/"ask", relation/"ask", arg1/"?", relation/"?"}
```

and the corresponding `kitchenAg` subscription would be

```
msg_name == "ask" && equals(arg1, "kitchen", "?")
```

Here `equals` is the Elvin subscription primitive for testing if an attribute has one of a sequence of values. In this case it tests if `arg1` has the value `"kitchen"` or `"?"`..

Finally, because the evaluation of the test part in Pedro subscriptions can use backtracking, we can use nested negation to implement forall tests. Suppose a client `fred` wanted to watch for party announcements of the form

Copyright © 2009 John Wiley & Sons, Ltd.

```
party_announcement(Invitees, FriendsOfHost)
```

in which `fred` is included in the invitee list and for which every invitee is a friend of the host. The subscription would be

```
subscribe(party_announcement(Invitees, FriendsOfHost),
          (member(fred, Invitees),
           not((member(M, Invitees),not(member(M, FriendsOfHost))))),
          ..)
```

We believe that, in Elvin, the information would need to be sent as a notification using strings for both the invitees and friends. It's not clear to us how the required subscription could be expressed.

5.2. Brief Comparison with ICM

Pedro peer-to-peer message handles are basically the same as those used in ICM except that Pedro allows variables in parts of the handles, making it more flexible for communicating with multiple clients. As with Elvin, ICM messages are not human readable - the ICM API is needed to encode/decode messages.

In a typical ICM application, there is one ICM server per machine and a message from one client to another on a different machine is sent to the local ICM server that forwards the message onto the ICM server of the destination machine. That server then forwards the message to the receiving client.

Pedro clients that wish to send messages to each other need to be connected to the same Pedro server.

5.3. Brief Comparison with xmlBlaster

xmlBlaster [18] is a publish/subscribe system that also provides peer-to-peer communication. Notifications are written in XML and subscriptions are made using XPath queries [19]. Peer-to-peer messaging is achieved via notifications of a particular form that include the address of the receiver. From the documentation provided on the xmlBlaster web site, it appears that the part of a notification that satisfies the XPath query is sent to the subscribing client. It also appears that the xmlBlaster provides persistent storage of previous notifications and that a new client subscription will pick up matching notifications from the persistent store. Notifications do not persist in Pedro.

5.4. Prolog for Implementing Pedro

Most modern Prologs have built-in support for TCP/IP so could be used to implement Pedro. Indeed, Pedro contains a Prolog-like compiler and a very cut-down WAM-like abstract machine. However, Pedro is much more lightweight than a typical Prolog implementation which contains a large collection of predicate definitions and other support, such as a heap garbage collection, that is not required in Pedro. There are also crucial differences between the Pedro and Prolog which need to be considered.

The first is the different semantics for some primitive predicates. For example, consider the `member/2` predicate in Pedro. The semantics is quite different from the semantics of the predicate as used in Prolog when it comes to lists with variable tails. When a `member/2` condition in a Pedro query tries to access a variable tail it fails. In Prolog it instantiates the tail. Consider the subscription

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

```
subscribe(lst(L),(member(X,L),nonvar(X)),..)
```

If the normal Prolog implementation of `member/2` was used, and this is tested against a notification `lst(L1)` where `L1` is a variable, the backtracking evaluation of the query would be non-terminating.

Furthermore, an arithmetic expression involving a variable or a non-number causes failure in Pedro but would typically throw an exception in Prolog. These problems would have to be overcome by giving Prolog definitions of the Pedro primitives.

Another issue is the question of occurs checking during unification. Consider the subscription query

```
X = [a|X], member(b,X)
```

with `X` an unbound variable. Without the occurs check (which does not allow a variable to be unified with a term containing the variable), the unification `X = [a|X]` succeeds binding X to a cyclic term representing an infinite list of `a` elements. This would lead to non terminating evaluation of the `member/2` condition, even with the Pedro implementation of `member/2`. Consequently, a Prolog implementation would need to make sure occurs checking is done on every unification.

Finally, if a Prolog without atom garbage collection was used to implement Pedro then eventually the atom table would overflow resulting in runtime failure. An implementation in Prolog that was restricted to use a fixed set of atoms would not be able to use arbitrary Prolog terms as notifications and subscription templates.

6. FUTURE WORK

The current version of Pedro supports multiple Pedro servers where clients can connect to more than one server. However, these servers are not federated as in Elvin. One area of future work is to look at what federation would have to offer for Pedro applications.

Pedro is an 'open' system - any client can send any notification. In some applications, such as intelligent buildings, we might want more security. One possible solution is to provide trusted Pedro servers that would only allow connections from specified clients. It would then be possible to run both trusted and untrusted Pedro servers and provide 'gatekeeper' clients that filter messages between the trusted and untrusted domains. Such gatekeepers could, for example, be rule-based Qu-Prolog clients.

Some applications can benefit from having each notification forwarded together with a time of receipt by Pedro. An associated time of arrival, paired with an arrival count to differentiate between notifications that arrive within the same timestamp interval, can be used by Pedro as a temporary unique notification identifier. This can then be used by Pedro to prevent the multiple forwarding of a notification to a client covered by more than one subscription from the client. The adding of Pedro time stamps to forwarded notifications is currently under consideration.

REFERENCES

1. Baldoni, R., M. Contenti, and A. Virgillito. The Evolution of Publish/Subscribe Communication Systems. Future Directions of Distributed Computing, Springer Verlag LNCS Vol. 2584, 2003.
2. Buschmann, F. et al, Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Wiley, 1996.

Copyright © 2009 John Wiley & Sons, Ltd.

3. Clark, K. L., Robinson, P. J., and Hagen, R,. Multi-threading and Message Communication in Qu-Prolog, *Theory and Practice of Logic Programming*, 1(3), pp 283-301, 2001.
4. Clark, K. L., Robinson, P. J., and Zappacosta-Amboldi S., Multi-threaded communicating agents in Qu-Prolog, in Computational Logic in Multi-agent systems, eds. F Toni and P. Torroni, LNAI 3900, Springer,2006.
5. Hohpe, G. and Woolf, B. Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions. Addison-Wesley, 2004.
6. Huhns, M. and Singh M., Ontologies for agents, IEEE Internet Computing,81-83, 1997.
7. T. Finin, Y. Labrou and J. Mayfield, KQML as an Agent Communication Language, *Software Agents*, (ed J. Bradshaw), AAAI/MIT Press, 1997
8. FIPA-ACL, Foundation for Intelligent Physical Agents, http://www.fipa.org/repository/aclspecs.html.
9. Aït-Kaci, H., *Warren's Abstract Machine — A Tutorial Reconstruction*, MIT Press, Cambridge, Massachusetts, 1991.
10. D. Kuokka and L. Harada, On using KQML for Matchmaking, 1st International Joint Conf. on Multi-agent Systems,pp 239-245, MIT Press, 1995.
11. David Luckham, The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison Wesley Professional, May 2002
12. F.G. McCabe, Interagent Communications Model, Technical Report, Fujitsu laboratories Ltd, 1999 (no longer available).
13. Muhl, G., Fiege, L., Pietzuch, P., Distributed Event-Based Systems, Springer, 2006.
14. B. Segall et al, Content Based Routing with Elvin4, Proceedings AUUG2K, Canberra, Australia, Downloadable from http://elvin.dstc.com/doc/papers/auug2k/auug2k.pdf, 2000.
15. J.A. Robinson, A Machine Oriented Logic Based on the Resolution Principle, Journal of the ACM,12 (23-41), 1965.
16. Reid G. Smith, The Contract Net Protocol: High-level communication and control in a distributed problem solver, in Readings in distributed artificial intelligence, eds. Alan H. Bond and Les Gasser, 357-366, Morgan Kaufmann, 1988.
17. Gerhard Weiss, ed. Multiagent Systems, A Modern Approach to Distributed Artificial Intelligence, MIT Press, 1999
18. xmlBlaster.org
http://www.xmlblaster.org/ [17 March 2009].
19. XML Path Language (XPath)
http://www.w3.org/TR/xpath/ [17 March 2009].

Copyright © 2009 John Wiley & Sons, Ltd.