SDTP — A Multilevel-Secure Distributed Transaction Processing System

Fred Gilham and David Shih
System Design Laboratory
SRI International
Menlo Park, CA
gilham, shih@sdl.sri.com

October 21, 1999

Abstract

In this paper we describe SDTP, a multilevel-secure distributed transaction-processing system that was written largely in Lisp, and two applications built on top of the SDTP system. We also discuss the experience of building the system.

We feel the system is of interest because

- It is moderately large.
- It attempts to implement and extend a significant published standard (the X/Open DTP standard).
- It uses a wide variety of facilities.
- It illustrates some of the advantages of using Lisp.

1 The SDTP Architecture

The X/Open DTP standard for distributed transaction processing consists of a protocol specification and a set of services that the components of the system must make available to other system components. SRI's DSA (Dependable System Architecture) group has used this standard as a testbed for applying our methods for verifying and extending formal descriptions of architectures.

Our most recent project in this area involved extending the X/Open DTP standard to incorporate multilevel security properties. This paper describes a prototype reference implementation that implements this extended standard, along with two applications built using the SDTP system [5].

1.1 X/Open DTP

The X/Open DTP standard [9, 11, 10, 12] is intended to standardize the interactions and communications between the components of the 3-tiered client-server model for distributed transaction processing. It allows multiple application programs to share heterogeneous resources provided by multiple resource managers (i.e. database managers, print managers etc.) and allows their work to be coordinated into global transactions.

A version of the X/Open architecture, shown in Figure 1, consists of three types of components—one application program (AP), one transaction manager (TM), and one or more resource managers (RMs). The boxes indicate the component interfaces, and the lines indicate the communications between them. The label TX indicates a complex connection and protocol defining communication between any application module and any transaction manager. This connection contains communication channels between functions that initialize and finalize transactions. Communication is always initiated by the application. A series of calls back and forth continues until communication is completed.

Similar complex connections exist between the application and every resource (the AR connection) and between the transaction manager and every resource manager (the XA connection). The XA connection provides communication for the well-known two-phase commit protocol that ensures the atomicity of transactions. Much of this activity can be concurrent, and many transactions may take place at once.

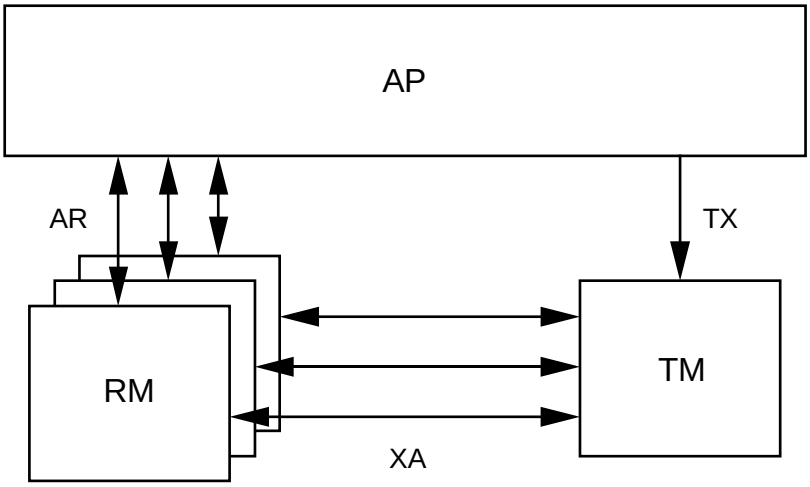

Figure 1: X/Open DTP Reference Architecture

The X/Open standard talks about these connections in terms of the services (TX, AR, XA) that each component must provide and the interface functions that implement these services. The TX service, also known as the Transaction Demarcation service, must be provided by the TM. The RM provides the AR service, which is the actual data storage interface to the RM. The AP makes use of the TX and AR services. The XA service consists of two subservices, one provided by the TM and the other provided by the RMs. The former, called the AX subservice or AXS, allows RMs to dynamically register and unregister themselves. The latter, called the XA subservice or XAS, exports the procedures the TM uses to coordinate the transactions. Both the TX and XA services are fully specified, albeit informally, while the implementation is allowed to use a custom set of functions for the AR service, allowing implementations to build custom resource managers. Table 1 gives a complete listing of the TX and XA services.

1.2 The Transaction Model

The X/Open DTP standard specifies a two-phase commit model to maintain a set of desirable transaction processing integrity properties known as the *ACID* properties. ACID is an acronym consisting of the first letters of the properties. These properties are

Atomicity The transaction is either executed completely or not at all.

Consistency A consistent transaction must take the database from one consistent state to another.

Isolation A transaction should appear to executed in isolation from all other transactions, as if it were the only transaction being executed by the system.

Durability Once a transaction has been committed, the changes it makes to the database must persist in the face of failures such as crashes.

The two-phase commit protocol is one method for maintaining the ACID properties in a distributed system. In our implementation it consists of the following steps.

- An application uses the TX:OPEN call to inform the transaction manager (TM) that it wishes to communicate with the resource managers (RMs) . A unique thread id (TID) gets generated (in our case, we use the TM to do it) and assigned to the application. The TID allows the RMs to correlate transactions with the clients on behalf of which it is performing them.
- The TM uses the AXS:OPEN call.to tell the resource managers (RMs) that an application will be contacting them.
- The application will do any necessary setup with the RMs. In our case, the application must authenticate itself to the RMs. The calls it uses are non-standard AR service calls.

Name	Description
TX:BEGIN	Start a new transaction
TX:CLOSE	Close the resource managers
TX:COMMIT	Complete a transaction normally
TX:INFO	Query the TM about the status of a transaction
TX:OPEN	Open the resource managers
TX:ROLLBACK	Abort a transaction
TX:SET-COMMIT-RETURN	Wait for commit completion or just for logging
TX:SET-TRANSACTION-CONTROL	Indicate 'chained' transactions
TX:SET-TRANSACTION-TIMEOUT	Set transaction time limit
AXS:REG	Let the RM register itself with the TM
AXS:UNREG	Let the RM unregister itself
XAS:CLOSE	Tell RM not to listen for connections any more
XAS:COMMIT	Tell RM to commit the transaction
XAS:END	Tell RM to end a transaction
XAS:OPEN	Inform RM that application is opening it
XAS:PREPARE	Ask RM if it can commit the current transaction
XAS:ROLLBACK	Tell RM to abort the transaction
XAS:START	Tell RM to start a transaction for a given application

Table 1: TX and XA services

- The application uses the TX:BEGIN call to tell the transaction manager it is starting a transaction.
- The application contacts the RMs using the TID generated by the TM, performing whatever operations constitute the transaction. Again, it uses AR calls to do this.
- The application uses the TX:COMMIT call to inform the TM that it has finished the transaction (alternatively, it can *rollback* the transaction using the TX:ROLLBACK call if an error occurs).
- The TM makes a XAS:PREPARE call to the RMs, asking them if they are prepared to commit the transaction (or, if the application has aborted the transaction, it informs the RMs using the XAS:ROLLBACK call.
- The RMs reply.
- If all the RMs reply in the affirmative within a given time, then the TM uses the XAS:COMMIT call to tell the RMs to commit the transaction. If not, it uses the XAS:ROLLBACK call to rollback the transaction.
- The application can call TX:BEGIN to start a new transaction, or TX:CLOSE to shut down the RMs.

1.3 Multilevel Security

A standard model of a multilevel security (MLS) policy is the Bell-LaPadula model [4]. Given a set of subjects each with an attached clearance level, and a set of objects each with an attached classification level, the model ensures that information does not flow downward in a security lattice by imposing the following requirements:

- **The Simple Security Property.** A subject is allowed a read access to an object only if the subject's clearance level is identical to or higher than the object's classification level in the lattice.
- **The * Property.** A subject is allowed a write access to an object only if the subject's clearance level is identical to or lower than the latter's classification level in the lattice.

The MLS policy regulates communication between the application and the resources. As such, it is primarily a property of the architectural structure. That is, it specifies that an application should not be allowed to connect to a resource manager that contains data for which it is not cleared.

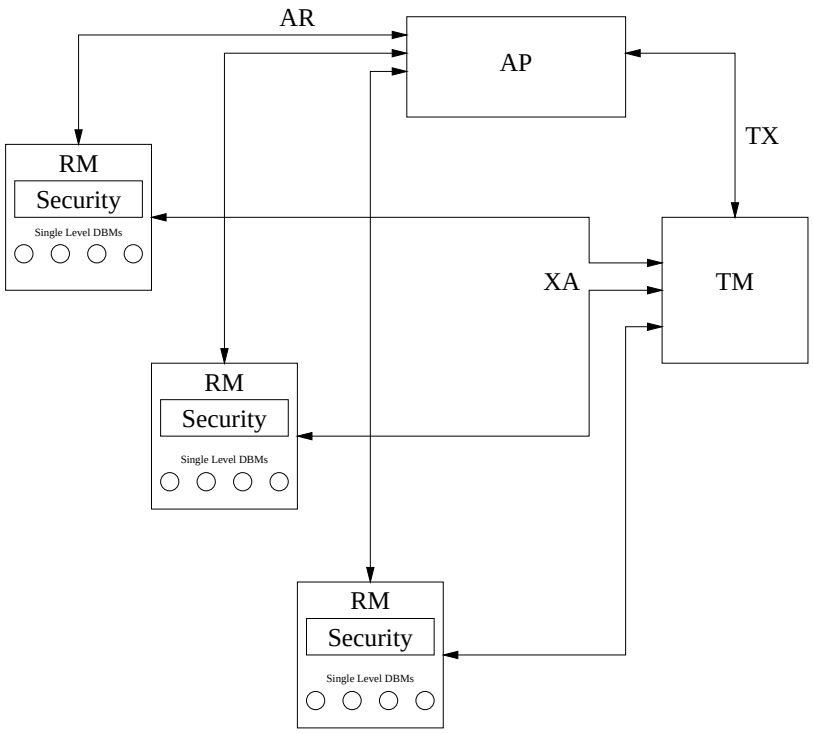

Figure 2: X/Open DTP Extended with Multilevel Security

1.4 Secure DTP

The SDTP implementation is based on a specification generated by formalizing the X/Open DTP specification in SADL [6], the Architecture Description Language used by our group, and then adding the MLS properties to the specification. The resulting specification was refined using provably correct transformations until an implementable specification was produced. The implemented architecture can be seen in Figure 2.

The refinement process by which this architecture was produced can be described as starting with communication channels that enforce the multilevel security policy, and then refining them into ordinary communication channels with the security policy enforcement implemented by a security manager. Finally, the security manager is distributed as security wrappers around each resource manager. The resource managers themselves are implemented as a collection of single-level database managers. The resulting specification preserves both the desired atomicity properties of X/Open DTP and the security properties mentioned above.

2 SDTP System Components

One goal of SDTP was a desire to be able to install it for our client and potentially other interested parties without the need to purchase expensive equipment and software licenses. For this reason, we tried to use freely available software whenever possible. It turned out to be possible to build the system with all major software components being in the public domain or having free redistribution licenses. The following components were important in building SDTP:

- **FreeBSD.** We chose the FreeBSD operating system because we were experienced with it and had found it reliable in the past. It runs on inexpensive Intel-x86 hardware and has a wide variety of freely available software, including database software and Lisp implementations, available through its 'ports' packaging system. In many cases, a desired software package can be installed by finding the appropriate 'ports' directory and issuing a 'make install' command. We found it more stable and less of a moving target than Linux, another freely available UNIX-like operating system.
- **CMU Common Lisp.** The CMU version of Common Lisp [1] is a high-quality Lisp implementation that is in the public domain. We have found its performance to be mostly competitive with, and occasionally better

than, commercial implementations. Its compiler also provides informative optimization notes that guide the programmer in making declarations that improve performance.

- **Common Lisp Bignum Facility.** To provide secure communications channels, we encrypted the data objects that were sent over them. To do this, we had to implement key exchange and encryption. Common Lisp's bignum facility turned out to be extremely convenient for this purpose. For example, Diffie-Hellman key exchange involved three functions that were each essentially one line of code (omitting declarations), along with a 'modpower' function ![8]:

```
;;; Modpower function from clmath package by Gerald Roylance.
(defun modpower (number exponent modulus)
  (declare (integer number exponent modulus))
  (do ((exp  exponent  (floor exp 2))
       (sqr  number (mod (* sqr sqr) modulus))
       (ans  1))
      ((zerop exp) ans)
    (declare (integer exp sqr ans))
    (if (oddp exp)
        (setq ans (mod (* ans sqr) modulus)))))

(defun compute-secret-key (dh-modulus)
  (declare (integer dh-modulus))
  (random dh-modulus (make-random-state t)))

(defun compute-public-key (base secret-key modulus)
  (declare (integer base secret-key modulus))
  (modpower base secret-key modulus))

(defun compute-common-key (remote-public-key
                           local-secret-key
                           modulus)
  (declare (integer remote-public-key
                    local-secret-key
                    modulus))
  (modpower remote-public-key local-secret-key modulus))
```

  While researching implementations we found a version written in the C programming language that took about 90 lines. Much of the C code involved processing a set of arrays, which in effect implemented a bignum capability. As a result, the connection between the C code and the algorithm being implemented was tenuous, while the Lisp code above is a direct implementation of the algorithm. The bignum facility was also useful in the actual encryption and decryption of data, allowing us to represent intermediate forms of the encrypted data as integers.

- **PostgreSQL.** We used the PostgreSQL[7] relational database manager to provide data storage management for SDTP. The choice of PostgreSQL was based on the fact that we had previous experience with it, it was easy to install and run, and it was licensed so as to allow us to distribute it conveniently.

  PostgreSQL comes with programmatic interfaces for several languages; unfortunately Lisp was not one of them. As a result we had to develop a Lisp interface to PostgreSQL.

  There are at least two ways to implement such an interface. One way is to talk directly to the PostgreSQL backend over the network. This involved creating packets and sending them to the backend. Since at the time we were not completely sure how this worked, we decided to take the second approach, which involved writing a foreign-function interface to the C version of the PosgreSQL client library.

  Like other Common Lisp implementations, CMU Common Lisp has a package that allows Lisp code to talk to code written in C. Interfacing to the PostgreSQL library involved writing a file of interface functions using

the CMUCL Foreign Function Interface. It turned out to be easy to write the code. There were a few opaque data structures, some enumerations for status returns, and a series of functions. The following is a sample of the code:

```
;; Opaque data structures.
(def-alien-type postgres-connection system-area-pointer)
(def-alien-type postgres-result system-area-pointer)

;; Status enumerations.
(def-alien-type connection-status (enum nil
                                        :connection-ok
                                        :connection-bad))

(def-alien-type exec-status (enum nil
                                  (:empty-query 0)
                                  :command-ok
                                  :tuples-ok
                                  :copy-out
                                  :copy-in
                                  :bad-response
                                  :nonfatal-error
                                 :fatal-error))

;; Some of the actual library interface functions.
(declaim (inline PQsetdbLogin))
(def-alien-routine "PQsetdbLogin" postgres-connection
  (pghost c-string)
  (pgport c-string)
  (pgoptions c-string)
  (pgtty c-string)
  (dbname c-string)
  (login c-string)
  (passwd c-string))

(declaim (inline PQexec))
(def-alien-routine "PQexec" postgres-result
  (connection postgres-connection)
  (command c-string))

(declaim (inline PQresultStatus))
(def-alien-routine "PQresultStatus" exec-status
  (result postgres-result))

(declaim (inline PQntuples))
(def-alien-routine "PQntuples" int
  (result postgres-result))

(declaim (inline PQnfields))
(def-alien-routine "PQnfields" int
  (result postgres-result))

;; etc.
```

A couple of problems arose from using this method. The main problem was due to an interaction between FreeBSD's object format and CMUCL. CMUCL is currently linked statically under FreeBSD, which means it

cannot use shared binary objects. The PostgreSQL C library is built using shared objects, for both its shared and static versions. (We think this is a bug but were unable to convince the maintainers to change it.) At any rate, we were forced to create by hand a library using static objects.

A similar problem involved the process of actually loading the library into the running Lisp. Since CMUCL running under FreeBSD uses an old technique of runtime linking, rather than the newer technique of dynamic loading, the C code library can be loaded only once per session; multiple loadings result in multiple definition errors.

- **Remote Procedure Call.** SDTP is implemented as a set of components and communication facilities. Our desire was to try to reflect current practice in implementing a DTP system. Thus, communication between the components is done using a Lisp-based Remote Procedure Call (RPC) mechanism.

  The components of SDTP employ a Lisp-based RPC mechanism for all communication except the communication with the database managers (which is done through the programmatic interface library discussed above). The RPC mechanism used is the Remote package provided by CMU Lisp. It is built on a lower-level socket based package called the Wire package. The Remote package allows calls like the following:

  ```
  (wire::remote-value rm-wire (xas:open tmid rmid flags))
  ```

  where `rm-wire` is a handle on the communication link over which the call is being made. This call causes the

  ```
  (xas:open tmid rmid flags)
  ```

  procedure call to be evaluated in the remote Lisp process referenced by the `rm-wire` handle.

  The Remote package has several limitations. First, it is limited in the data objects it can actually send between the communicating processes. It lacks a fully general external data representation (XDR) mechanism. On the other hand, it allows a remote Lisp process to refer to a local data structure by passing a token called a remote object. The remote process can, without actually modifying the data object, effectively pass it back as a return value or pass it as a parameter to a call-back procedure it invokes in the local process. Communications channels created by the Remote package are bidirectional; a remote Lisp process can invoke procedures in the local process as well as the other way around. If a process needs to actually send data objects that are not supported by the Remote package, it must convert them to a string format and send them as strings, whereupon the remote process must convert them back into data objects. In practice, limitations of data object types did not arise in this system.

  Another limitation was the fact that the Remote package provided no access control. A process making remote procedure calls can invoke any procedure in the remote Lisp process, if it knows the package and procedure name. This creates a security hole; at the very least a malicious client could invoke a denial-of-service attack by remotely calling the `(quit)` procedure. It would be fairly straightforward to create a version of the Remote package that required each process to register procedures that it would allow to be invoked remotely; such an extension was not made in this implementation but will probably be done in the future.

- **Graphical User Interface Toolkit.** At first, SDTP did not have a GUI. In the form we originally demonstrated, it had a graphical display to show the interactions between the components (see Figure 3). To make the demonstration application more usable, we decided to give it a GUI. All the GUI work in SDTP, including the component-interaction graph, was done using the Garnet [2] toolkit.

  The Garnet toolkit, besides being freely distributable, had good performance and a relatively small footprint in CMUCL. An x86 Lisp image with Garnet is about 18 MB, while the image without Garnet is about 14 MB (CMUCL's images are considerably larger than current commercial versions). This compares to a size of around 28 MB for a CLOS-based toolkit such as XIT/CLUE [3].

  Originally we were concerned about the fact that Garnet did not use CLOS as its object system; instead it uses KR, a relatively simple prototype-instance object system with constraints. However, with experience we came to feel that KR's approach was a good match for GUI development, and its (relatively) small size and low overhead served us well when we installed the system on laptops and other resource-limited machines. At one point we had several Lisp processes, including a Garnet process, running on a 32 MB 200 MHz Pentium

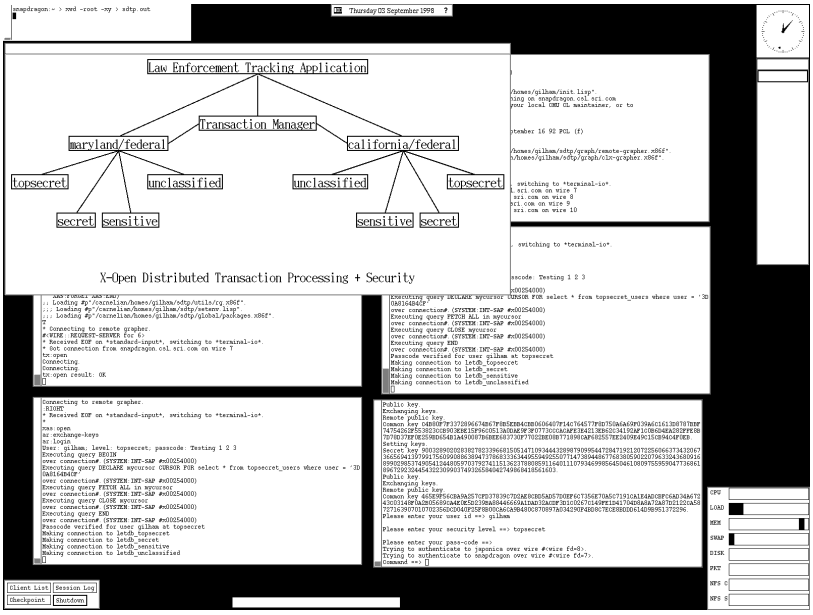

Figure 3: SDTP with Component Interaction Graph

machine. Performance, while not sterling, was adequate. CMUCL uses the PCL version of CLOS. Experience with the XIT toolkit showed that it caused PCL to spend a lot of time doing runtime compilation, especially when it loaded files, while Garnet did little or no runtime compilation.

Among Garnet's important features that we used heavily were objects known as 'interactors' that deal with various types of user input. Interactors can be attached to different graphical objects, thus almost completely decoupling the look of the object from the way the object responds to user interaction. As a result of the use of interactor objects, Garnet allows a programming style that minimizes the use of callbacks. Instead of an event-driven model where the programmer writes procedures that are given control in response to user actions, the programmer can use a more functional approach. We ended up in effect calling the GUI as a function that returns the results of the user's activity. The messy event handling and synchronization is kept neatly behind the scenes by the interactor mechanism.

Garnet provides two toolkits with different appearances. We used both; for the component-interaction graph we used Garnet's native toolkit, while for the application GUIs we used the second toolkit that provides a Motif-like appearance.

3 Applications

We built two applications on top of SDTP. The first is a demonstration application used to show that SDTP works and has the desired security properties. The second is intended as the genesis of a real-world application that would serve as an intelligent post-processor to a computer intrusion-detection system. Figure 4 illustrates the GUI style of both applications.

3.1 Cooperating Law Enforcement Databases

As a test application we built a multilevel-secure law-enforcement-tracking database, LET-DB. This was inspired by an FBI system called "FOIMS" (Field Office Information Management System) though of course it bears no relationship to the actual working of that system.

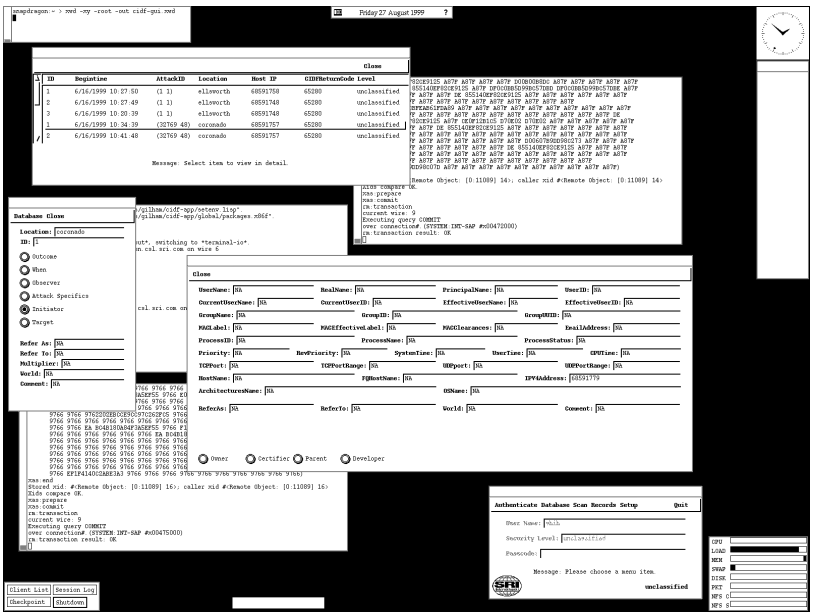

Figure 4: SDTP Application GUI

Our intent in building this application was to demonstrate multilevel security in a database system that was both distributed and replicated. For this reason we decided to distribute the information for state investigations and replicate the federal information. The motivation for this was that each state would tend to query and update its own information the vast majority of the time and so it would be more efficient to keep that information on local resource managers and allow remote queries; on the other hand, federal information would probably be queried by many states and so it would be more efficient to replicate that information across the resource managers belonging to the states.

An example of a table that depended on multilevel security was the agent table. This table contained a unique ID number for each agent as well as the agent's name. The ID number was needed at all security levels, both to keep track of agent workloads and to ensure that each investigation had a valid agent assigned to it. However, the agent's name was considered top secret and so it was available only when the user making the query was authenticated at the top secret security level.

Other examples of multilevel tables were the investigation tables themselves. Each investigation was classified at a particular security level; even the fact that an investigation was being conducted on a particular individual might itself be sensitive information. For this reason, investigations that might have, for example, important political implications were classified top secret, while investigations that had resulted in court cases would ordinarily be public knowledge and therefore unclassified.

Our demonstration implementation contained resource managers for two states, each of which also contained a replica of the federal portion of the database. The application was capable of querying, updating, and adding information to the relevant tables.

3.2 Intrusion Detection Application

The second application built on SDTP is the Intrusion Detection Application. One of the authors of this paper, David Shih, was hired as a summer intern to write this application. He has completed his first year as a Computer Science major at the University of California at Berkeley. His most significant programming experience was his first-semester computer science programming course, taught using Scheme. Thus he, like the other author, was writing his first major Lisp program.

The Intrusion Detection Application turned out to be larger and more complex than the Law Enforcement Track-

ing application. Like the LET-DB application, the IRDB (Intrusion Recording Database) also manages multilevel secure databases, this time containing computer intrusion data. This application receives, from one or more secure sources, CISL (Common Intrusion Specification Language) data. CISL is a proposed standard for representing and exchanging computer intrusion data. A complete draft on CISL and its syntax can be found at

`http://gost.isi.edu/projects/crisis/cidf/cisl_current.txt.`

Conveniently, CISL uses an s-expression format for its external representation.

In our case, the application receives CISL data describing intrusion attempts against machines located at various military installations. (For example, our current demonstration pretends to receive data from two, Ellsworth AFB and NAB Coronado). The application currently reads CISL data, classifies each entry, and inserts the entry into the proper database. A user can then query the database, retrieve CISL entries, and look through an entry in detail in a more user-friendly environment. Eventually the application will perform intelligent pattern-matching across the database entries to check for clues that would suggest a serious breach attempt against the network

Represented as s-expressions in a tree structure, CISL attack entries express specific information about intrusion attempts by describing the verb/action (in this case, Attack) with a sublayer of role and adverb entries. These describe the "players" involved and attributes of the verb, respectively. Beneath role entries are attribute entries describing role attributes such as the owner or developer of that particular player. All these entries—verb, role, adverb, and attribute—contain what are known as atomic entries. An atomic entry contains the actual values which describe the entry within which it is enclosed. So a sample generic structure for a CISL entry could look something like

```
(Attack                          ; verb SID (semantic identifier)
  (World Redhat-5.1)             ; atomic SID and value
  (Outcome                       ; adverb SID
    (atomSID2    value2)
    (atomSID3    value3))
  (Observer                      ; role SID
    (atomSID4    value4)
    (atomSID5    value5)
    (Owner                       ; attribute SID
      (atomSID6 value6))))
```

Currently, to store data, a batch-mode data-storage application simulates the remote data streams that we plan to use. Since the CISL data we read has no provision for classification of records, the batch-mode program reads the input and classifies the data. To do this, it performs a tree search for the atomic field containing the IP of the targeted machine. From the IP, it determines the record's security classification and location. The intuition behind this is that there may be sensitive information exposed by an intrusion attempt—such as an unaddressed vulnerability of a classified machine, or even the existence of said machine—that users below a certain security level should not have access to. By classifying records by target IPs, records about these machines can be hidden away at the proper level. Eventually, however, instead of receiving data from a single mixed data stream and having the application organize the data into different classification levels, the application will receive information from pre-classified data streams, allowing the application to simply tag the data with the security level associated with the stream from which it was received, and insert it into the proper database.

As mentioned above, input into IRDB can be given only through a secure data stream containing raw CISL data. The application then has to parse the CISL s-expressions into string entries that can be stored into the PostgreSQL database. Because CISL was designed to be extensible, insertion of CISL s-expressions into a relational database becomes a chore because there is no guaranteed uniformity between one CISL entry and the next. Atomic fields and sub-rows that existed in one CISL construct may not exist in another. As a result, the application has to make sure that existing atomic fields that were not reported in a CISL entry are filled in with some kind of no-data identifier and that role entries and adverb entries to the attack entry have a similar procedure applied to them and to their attribute entries. These new "completed" rows, which now all contain the same number of entries, are then inserted into their respective tables.

Consequently, because of the layered makeup and the potentially large size of a CISL entry, it seemed more sensible to break up the large CISL entry into its verb, role and adverb entries and insert each of those entries into

its own separate table rather than create a table with more than 200 columns. To maintain uniformity across the multiple tables, the application tags each subentry with a unique ID number. In dealing with the attribute entries, each role entry that allows for attribute entries has columns in its table reserved for each attribute. The application then simply inserts attribute entries in their s-expression form as a whole into the column within the role table to which they belong. Since all attribute entries do actually have the same number and type of fields, it didn't seem necessary to create sixteen extra tables just to store them. Instead, we use the process described above and employ a kind of lazy procedure to format the attribute entry from s-expression to string when the user actually asks that a particular attribute item be shown in detail after the query results are displayed during lookup.

4 Experiences

Both authors of the system were writing their first major Lisp program. One has more than fifteen years of programming experience in everything from assembly language and microprocessor BASIC to C; the other was writing his first major program in any language. Both programmers were pleased with Lisp as part of a development infrastructure.

A major feature of our development experience was the interactive programming environment. We used the Ilisp mode for Emacs, which gave convenient access to the interactive features of Lisp.

This took getting used to. One of us, used to the C batch-mode development environment, found himself constantly killing off the Lisp process and restarting it every time he made a change. Eventually, it dawned on him that it wasn't necessary to do this. It was a revelation when he modified the text of a function and, using the Ilisp `(compile-defun-and-go)` command, added it to the currently running system and saw the change take immediate effect. This was especially helpful because, upon restarting the system, it often took a minute or so to get back to the point where the change could be tested.

Another instance where the interactivity of Lisp was valuable was at the conference where the system was first demonstrated. The demonstrator made a typo that caught a bug in the application. This resulted in the application entering the Lisp debugger with a segmentation violation error. (This error indicates that the process has made an invalid memory access, for example, to an area of memory that is not currently part of its valid address space.) The demonstrator quit the debugger, getting back to the Lisp toplevel, and then re-entered the main loop of the application and picked up where he left off. The total time consumed by the crash was about ten seconds, and it appeared that the person watching the demo didn't actually know that the system had crashed. This is in contrast to the chaos that would have resulted if an equivalent C program had gotten a segmentation violation.

The component-visualization grapher turned out to be an extremely effective part of the demo, helping people understand the interactions between the components of the system. It gave a real-time view of what was going on in the system and made it clear how the multilevel security aspects operated. This grapher was written using Garnet and was easy to implement and test.

We have already described the major enhancements and additions we made to the system—adding a GUI to the original application and writing a second application. As we mentioned above, the task of adding a GUI to a command-line-oriented application was simplified by the programming style allowed by the Garnet toolkit. Instead of having to turn the program flow inside out to conform to the event-driven model of typical X-based GUI toolkits, we were able to use a functional approach that retained the original program flow. Below is part of the main loop of the application. The main change from the original text-based UI is the use of the calls to the SG (SDTP-GUI) package instead of using text-based alternatives.

```
(defun do-application ()
(loop
 (let* ((user (if *resource-managers*
                  (rm-data-user (car *resource-managers*))
                nil))
        (command (sg:application-interface user
                                           (if (minusp *current-level*)
                                               ""
                                             (level-number-to-name *current-level*)))))

   (sg:toplevel-message "Please choose a menu item.")
```

```
(case command

  (:login
   (setup-tm-connection)
   (setup-rms)
   (login))

 ;; Logout and close GUI.
  (:quit
   (shutdown)
   (sg:force-quit)
   (return-from do-application nil))

   ...etc...
```

In general we found that the use of Lisp made programming the system a pleasant experience. We were able to use a wide variety of solutions to the various problems the system posed without fear that debugging new approaches would be a tedious, time-consuming process. Both developers learned new programming techniques as a result of building this system. Most important, our superiors and clients were impressed with the results of what we produced.

References

[1] CMU Common Lisp User's Manual. Technical Report CMU-CS-92-161, School of Computer Science, Carnegie Mellon University, July 1992.

[2] Brad A. Meyers et. al. The Garnet Reference Manuals. Technical Report CMU-CS-90-117-R5, School of Computer Science, Carnagie Mellon University, December 1994.

[3] Jürgen Herczeg, Hubertus Hohl, and Matthias Ressel. XIT — *The X User Interface Toolkit, Programming and Reference Manual.* Research Group DRUID, Department of Computer Science, University of Stuttgart, February 1995.

[4] C.E. Landwehr. Formal Models for Computer Security. *ACM Computing Survey*, 13(3):247–278, September 1981.

[5] M. Moriconi, X. Qian, R. Riemenschneider, and L. Gong. Secure software architectures. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.

[6] Mark Moriconi and R. A. Riemenschneider. Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies. Technical report, Computer Science Laboratory, SRI International, September 1996.

[7] The PostgreSQL Development Team. *PostgreSQL User's Guide*, 1999.

[8] Gerald Roylance. Some Scientific Subroutines in LISP. Technical Report AIM-774, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, September 1984.

[9] X/Open Company Ltd., Apex Plaza, Forbury Road, Reading, Berkshire RGI 1AX, U.K. *Distributed Transaction Processing: The XA Specification*, June 1991.

[10] X/Open Company Ltd., Apex Plaza, Forbury Road, Reading, Berkshire RGI 1AX, U.K. *Distributed Transaction Processing: The Peer-to-Peer Specification*, December 1992.

[11] X/Open Company Ltd., Apex Plaza, Forbury Road, Reading, Berkshire RGI 1AX, U.K. *Distributed Transaction Processing: The TX (Transaction Demarcation) Specification*, November 1992.

[12] X/Open Company Ltd., Apex Plaza, Forbury Road, Reading, Berkshire RGI 1AX, U.K. *Distributed Transaction Processing: Reference Model, Version 2*, November 1993.