Reliable Distributed Computing with Avalon/Common Lisp

Stewart M. Clamen, Linda D. Leibengood, Scott M. Nettles, and Jeannette M. Wing

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Avalon is a set of linguistic constructs designed to give programmers explicit control over transaction-based processing of atomic objects for fault-tolerant applications. These constructs, designed as extensions to familiar programming languages such as C++ and Common Lisp, are tailored for each base language so the syntax and spirit of each language are maintained. We present here an overview of these novel aspects of Avalon/Common Lisp: (1) support for *remote evaluation* through a new **evaluator** data type; (2) a generalization of the traditional client/server model of computation, allowing clients to extend server interfaces and server writers to hide aspects of distribution, such as caching, from clients; (3) support for *failure atomicity* through automatic commit and abort processing of transactions; and (4) support for *persistence* through automatic crash recovery of atomic data. These capabilities provide programmers with the flexibility to exploit the semantics of an application to enhance its reliability and efficiency. Avalon/Common Lisp runs on IBM RT's on the Mach operating system. Though the design of Avalon/Common Lisp exploits some of the features of Common Lisp, e.g., its packaging mechanism, all of the constructs are applicable to any Lisp-like language.

1. Introduction

Large networks of computers supporting both local and distributed processing are now commonplace. Application programs running in these environments concurrently access shared, distributed, and possibly replicated data. Examples of such applications include electronic banking, library search and retrieval systems, nation-wide electronic mail systems, and overnight-package delivery systems. Such applications must be designed to cope with failures and concurrency, ensuring that the data they manage remain *consistent*, that is, are neither lost nor corrupted, and *available*, that is, accessible even in the presence of failures such as site crashes and network partitions.

A widely-accepted technique for preserving consistency in the presence of failures and concurrency is to organize computations as sequential processes called *transactions*. Transactions must satisfy three properties: serializability, failure atomicity, and persistence. *Serializability* means that transactions appear to execute in some serial order. *Failure atomicity* ("all-or-nothing") means that a transaction either succeeds completely and *commits*, or *aborts* and has no effect. *Persistence* means that the effects of a committed transaction survive failures. We use the term *atomic* to stand for all three properties.

Although transactions are widely used in the database community, demonstrating that they can be a foundation for general purpose distributed systems remains a challenge and is currently of active interest. Appropriate programming language support for application programmers would greatly enhance the usability and thus, generality, of such systems.

Avalon is a set of linguistic constructs designed as extensions to familiar high-level programming languages such as C++ [25] and Common Lisp [16]. The extensions are tailored for each base language, so the syntax and spirit of each language are maintained. The constructs include new encapsulation and abstraction mechanisms, as well as support for concurrency and recovery. The decision to extend existing languages rather than to invent a new language was based on pragmatic considerations. We felt we could focus more effectively on the new and interesting issues such as reliability if we did not have to redesign or reimplement basic language features, and we felt that building on top of widely-used and widely-available languages would facilitate the use of Avalon outside our own research group.

This paper presents an overview of some of the more novel aspects of Avalon/Common Lisp. The distinguishing characteristic of Avalon/Common Lisp, in contrast to Avalon/C++ [6] and other transaction-based distributed programming languages (see Section 6), is its support for *remote evaluation* [23]. Lisp's treatment of code as data provides a natural and easy way to implement remote evaluation since we simply transmit code, as well as data, between clients and servers. Moreover, we exploit remote evaluation to extend and generalize the traditional client/server model of distributed computing. Thus, the programmer gains more flexibility in structuring an application, while often simultaneously improving its performance.

We have implemented the Avalon/Common Lisp constructs presented herein on top of Camelot [21], a distributed transaction management system (written in C) built at Carnegie Mellon. Camelot provides low-level facilities like lock management, two-phase commit protocols, and logging to stable storage.

CH2854-8/90/0000/0169$01.00 © 1990 IEEE

The particular extensions we designed for Common Lisp are applicable to any Lisp-like language, though for concreteness, all our examples will be expressed in Avalon/Common Lisp. We assume the reader has a reading knowledge of Common Lisp.

In Section 2 we give an overview of Avalon/Common Lisp's model of computation and program structure as they relate to distribution, persistence, and concurrency. Sections 3 and 4 explain the novel features of Avalon/Common Lisp related to distribution, in particular remote evaluation and our generalization of the traditional client/server model. Section 5 illustrates features of Avalon/Common Lisp related to persistence. Section 6 compares Avalon/Common Lisp with other transaction-based, distributed programming languages and closes with a summary of our current status.

2. Overview

Distribution

An Avalon/Common Lisp computation executes over a distributed set of *evaluators* (Figure 1), each of which is a distinct Lisp process. An evaluator resides at a single physical site, but each site may be home to multiple evaluators. A user starts a computation at an *initiating* evaluator, which may communicate with other *remote* evaluators. To a first approximation, evaluators communicate through remote procedure calls with call-by-value semantics. The dotted lines in the figure indicate possible call paths between evaluators.

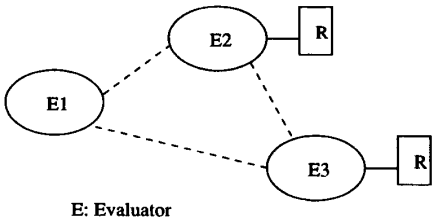

Figure 1: Model of Distributed Evaluators and Recoverable Storage

As in Common Lisp, an Avalon/Common Lisp program consists of a set of *packages*. Each evaluator is host to one or more packages. We map the standard client/server model of distributed computing onto our more general architecture as follows: We put a client's code in one package and execute it on the initiating evaluator, and for each server, we put its code in a separate package and execute it on a remote evaluator.

Section 3 will explain how we extend this standard client/server model by using remote evaluation in combination with the feature that an evaluator can be host to multiple packages. The combination frees us from the above one-to-one correspondences between client code (or server code) and a package, and between a client process (or server process) and an evaluator. In short, in our full extended client/server model, client code can cross evaluator boundaries, can be split into more than one package, or can coexist with server code at the same evaluator. Similar remarks hold for server code.

Persistence

Since Avalon/Common Lisp provides *transactions*, we need to provide a way to support failure atomicity and persistence. When a crash occurs, we need to recover the state of the system to some previously saved consistent state, one that reflects all changes performed by all committed transactions.

Each evaluator has access to at most one private *recoverable store* (see Figure 1), which itself is managed by a separate process.[1] Normally, there would be no recoverable store associated with the evaluator where the client code resides, but there would be one per evaluator that is host to a server.

At the programming language level, each server package encapsulates a set of *object bindings* and exports a set of *functions*. Each object binding is a mapping between a symbol and an object. A binding can be declared to be *persistent*; otherwise it is considered to be *volatile*. Persistent bindings (and the objects to which they map) are allocated from recoverable store; hence, persistent bindings survive crashes, while volatile ones do not. By convention, a server's functions should provide the only means for a client and other servers to gain access to the server's object bindings, and thus its recoverable objects.

It makes sense to access recoverable objects only when executing a transaction so Avalon/Common Lisp provides control primitives to begin, commit and abort transactions. Section 5 shows a use of these primitives.

Concurrency

Avalon/Common Lisp supports concurrent transactions ("heavyweight" processes), but no concurrency within a transaction. Serializability of transactions is guaranteed by using standard two-phase read/write locks on objects [8]. A transaction holds its locks until it commits or aborts.

Since Common Lisp does not support multiple threads of control, in particular "light-weight" processes as in C Threads [5], we have a simpler model of computation with respect to concurrency than that for other languages such as Avalon/C++. Specifically, only one thread of control executes within an evaluator at once. For example, suppose two clients each make a request at a single server. The (server's)

[1]Each recoverable storage manager is a C process since we currently use Camelot's implementation of recoverable storage; hence, each of our Lisp processes communicates with a C process whenever recoverable storage is accessed.

evaluator processes these two requests serially. On behalf of the first request, it accesses the recoverable store, acquires appropriate read or write locks, and returns appropriate result values. The evaluator then services the second request. If the second request creates a lock conflict, the (server's) evaluator blocks until the lock is freed. Lock conflicts can arise because locks are released as transactions complete, not when function calls return.

Avalon/Common Lisp supports nested transactions, but, again because of the limited kind of concurrency we can support in Common Lisp, each transaction can have at most one active child transaction. A transaction commits only if its child has committed or aborted; a transaction that aborts aborts its child. A transaction's effects become persistent only when it commits at the top level.

The most interesting and novel aspects of Avalon/Common Lisp relate to its way of handling distributed computing, and not persistent storage or concurrency. Thus, the next two sections will focus on the issues related to distribution: remote evaluation and the extended client/server model.

3. Remote Evaluation

3.1. Example Uses of Extensions

Suppose, for simplicity, there are two evaluators, one *local* and one *remote*, where the local evaluator might be the initiating evaluator for some computation. The following expressions:

```
(let ((a 123) (b 45)) (+ a b))
(let ((a 123) (b 45)) (remote (+ a b)))
(let ((a 123) (b 45))
     (remote (+ (local a) (local b))))
```

all return the same value to the user, namely the number `168`. Given that the function + refers to the built-in generic addition function, all three expressions have the same semantic meaning. How they differ is where the various subexpressions are evaluated.

In the first expression, all computation (new binding creation, variable lookup, function application) occurs on the local evaluator.

In the second expression, the creation of bindings for `a` and `b` occurs on the local evaluator, while the `remote` special form directs the evaluation of the `(+ a b)` to be performed on the remote evaluator. The lexical environment, containing the local bindings for `a` and `b`, is transmitted along with the expression `(+ a b)` to the remote evaluator.

The evaluation of the third expression occurs similarly to the second, except that the evaluations of the expressions `a` and `b` (within the `(+ (local a) (local b))` expression) are performed back on the local host. Since + is already defined on the remote evaluator, this process is equivalent to a traditional remote procedure call (RPC), where the arguments (and not the actual function) are evaluated locally and then transmitted to a remote server for application.

3.2. New Functions, Special Variables, and Macros

As an extension, Avalon/Common Lisp provides one new data type, the **evaluator**, two new special variables, `*remote-evaluator*` and `*local-evaluator*`, and a small number of new special forms, the most important of which are `remote` and `local`. Intuitively, the two forms are used to translate the thread of computation from one evaluator to another, e.g., from the designated local evaluator to some remote evaluator. Below we give the meaning of each in the style of the Common Lisp manual [16].

`make-evaluator` *string* [*Function*]

This function finds and returns the evaluator whose name is specified by the string argument. If none exists, it builds and returns a new evaluator object. Evaluators are *first-class* objects: one can store an evaluator away in other data structures, perform remote evaluations on it at some future time, and transmit them.

`*remote-evaluator*` [*Variable*]

This special variable names the evaluator used to evaluate expressions of the form `(remote` *expr*`)`. On an initiating evaluator, it is bound by default to the initiating evaluator itself until the user changes it to point to some other (remote) evaluator. On a remote evaluator, it is bound by default to the remote evaluator itself. If desired, the programmer can explicitly reset this binding dynamically.

`*local-evaluator*` [*Variable*]

This special variable names the evaluator used to evaluate expressions of the form `(local` *expr*`)`. In the case of an initiating evaluator, it is normally unbound. In the case of a remote evaluator, it is bound by default to the evaluator from which the `remote` was called. If desired, the programmer can explicitly reset this binding dynamically.

`remote` *expr* `&optional` *evaluator* [*Macro*]

This special form's semantics is identical to `identity` except that: (1) The actual computation is performed by the evaluator bound to `*remote-evaluator*` (or to the evaluator specified as the optional argument) with the same lexical environment as the current

evaluator, but a different current package and dynamic state; and (2) the object returned is a *copy* of the result, as opposed to the result object itself. Even in the case where the evaluator bound to `*remote-evaluator*` is specified to be or defaults to the current evaluator, a copy of the resulting object is returned.

Since the process for transmitting data from one evaluator to another necessitates creating copies of objects, mutable objects[2] are not `eq` to their remotely referenced analogues. This is the primary incompatibility introduced by the use of `remote` expressions in a program. Despite the loss of identity, we still preserve sharing of common substructures among transmitted objects, so that values that are comparable on one evaluator are still comparable on another. Hence, we have:

```
(let* ((a (the (not (or number symbol character))
              <arbitrary lisp object>))
       (b a))

  (eq a (remote a))                  ⇒ nil
  (remote (eq a (remote a)))         ⇒ nil

  (equalp a (remote a))              ⇒ t
  (remote (equalp a (remote a)))     ⇒ t

  (remote (eq a b))                  ⇒ t
  (eq (remote a) (remote b))         ⇒ unspecified
```

Here an object that is neither a number, symbol, nor character is locally bound to `a` and `b`. The first two comparisons return `nil` since the object bound to `a` and its copy are different objects, regardless of where the comparison is evaluated. The next two comparisons return `t` because the values of `a`'s object and its copy are the same. The next comparison shows that remotely comparing the identities of `a` and `b` is identical to comparing them locally. Finally, the last comparison shows that while remote and local copies are not identical, the results of different `remote` calls to the same evaluator may return the same object.

`local` *expr* [*Macro*]

This special form has meaning only when evaluated dynamically within a `remote` expression. Its semantics is identical to `identity` except that: (1) Computation occurs at the evaluator specified in `*local-evaluator*`; normally, this is the evaluator where the most dynamically immediate `remote` expression was evaluated; and (2) the object returned is a copy of the object, instead of the object itself.

[2]In Common Lisp, all objects, except for numbers, characters and symbols, are mutable.

Avalon/Common Lisp gives the programmer the flexibility to redirect the thread of computation, if desired, by using the optional parameter to `remote`, or by explicitly setting `*remote-evaluator*` to an evaluator different from the default. Hence, the user can make *third-party calls*, i.e., calls by one remote evaluator to another evaluator. Third-party calls would be common when one server calls another server on behalf of the original computation performed for the client. The calling evaluator is then defined to be the local evaluator and the third evaluator to be the remote evaluator. For example, in Figure 1, if E1 remotely calls E2 which then remotely calls E3, then E3's `*local evaluator*` is automatically set to E2 and its `*remote-evaluator*`, to E3.

Note that since special variables can be set dynamically, they need not reflect the call chain, though normally they would. In the previous scenario, for example, if E3's `*local-evaluator*` is explicitly reset to E1, then `local(...)` expressions would be evaluated at E1, not E2, even though E2 made the remote call to E3. Results are still returned to the evaluator that initiated the `remote` call; hence they would be returned to E2, not E1.

3.3. Abstract Interpreter

Figure 2 shows a simplified abstract interpreter, giving a more formal semantics to the evaluation of the special forms, `remote` and `local`. It does not handle the case of preserving (remote) side effects on shared, mutable objects.

We first define a `dynamic-state` to include the (current) lexical environment, control-related tags and labels, and names of the local and remote evaluators. The lexical environment includes both local variable and local function bindings. We define an `evaluator` to be a name and a set of packages.

To see what `eval` does, we first explain what the helping function `handle-remote` does. It takes four arguments: the expression being evaluated; a dynamic state that includes some lexical environment; and two evaluators, one to indicate where `local` expressions are to be evaluated and one to indicate where `remote` expressions are to be evaluated. A new dynamic state is created and used as the state in which the argument expression is evaluated. The `deep-copy` function preserves internal sharing of objects. It is similar to the `read` of a `print` on printable Common Lisp objects. The recursive calls to `eval` and `deep-copy` ensure that expressions with nested `remote`'s and `local`'s are handled properly.

The `eval` function itself takes three arguments, the expression being evaluated, a dynamic state that includes some lexical environment, and an evaluator. If the expression to be evaluated is a `remote` then first a check is made to see if a specific evaluator is bound to the optional argument in the `remote` call; if not, then the

```
(defstruct dynamic-state
  lexical-env
  catch-tags
  labels
  local-evalr
  remote-evalr
  )

(defstruct evaluator
  name
  packages
  )

(defun eval (expr state evalr)
  (case expr
        ;; other cases ...

        (remote
          (handle-remote (remote-body expr)
                         state evalr
                         (or (remote-evalr expr)
                             (dynamic-state-remote-evalr evalr))))
        (local
          (handle-remote (remote-body expr)
                         state evalr
                         (dynamic-state-local-evalr evalr)))))

(defun handle-remote (expr state oevalr nevalr)
   (deep-copy
     (eval
        expr
        (make-dynamic-state
          :lexical-env  (deep-copy (dynamic-state-lexical-env state))
          :catch-tags   (dynamic-state-catch-tags state)
          :labels       (dynamic-state-labels state)
          :local-evalr  oevalr
          :remote-evalr nevalr
          )
        nevalr)))
```

Figure 2: Abstract Interpreter for Handling Remote Evaluation

remote evaluator bound in the dynamic state is passed as the new remote evaluator to `handle-remote`. Handling a `local` is simpler; the local evaluator bound in the dynamic state is passed as the new remote evaluator to `handle-remote`.

3.4. More Examples

The environment passed as part of a `remote` call does not include the Common Lisp "special" (global) bindings. In the following example:

```
(defvar a 123)
(let ((b 45))
    (remote (+ (local a) b)))
```

an explicit call *back* to the initiating evaluator (using `local`) is required in order to ensure that the special value of `a` is retrieved; otherwise, the global binding of `a` on the remote evaluator would be used. Note that the default binding of `*local-evaluator*` will cause `local` to direct a computation back to its originating evaluator.

Since one of Avalon's design goals is to minimize interference with the target language's semantics, nearly all Common Lisp expressions can be "wrapped in" a `remote` to give the desired and expected effects. The lambda expression below is transmitted to the remote evaluator along with its argument for evaluation, illustrating that even procedural objects are permissible within `remote` expressions:

```
(remote ((lambda (x) (* x x)) 4))
```

We also support the remote application of recursively defined functions such as:

```
(labels ((fact (n) (if (< n 2)
                       1 (* n (fact (- n 1))))))
  (remote (fact 20)))
```

since the current lexical environment is transmitted along with the expression. During the evaluation of the above code, the recursive function `fact`, bound in the lexical environment, is applied to `20` on the remote evaluator, and the result is transmitted back to the local evaluator.

The effects of mutating operations in the lexical environment are preserved across evaluator boundaries. For example, the following returns `10`:

```
(let ((a 5))
  (remote (setq a 10))
  a)
```

We also handle exits, both local and dynamic, transparently. The result below will be `12`, just as if the `remote` call had never existed:

```
(block tag
  (remote (+ 9 (return-from tag 12))))
```

Likewise with the following, the result is also `12`:

```
(progn
  (remote (defun add9 (x)
                (+ x (throw 'foo 12))))
  (catch 'foo (remote (add9 1))))
```

3.5. Transmission of Objects

Avalon/Common Lisp supports transmission of all Common Lisp readable types. A type is *readable* if all its instances can be created through the Common Lisp reader using the type's default print representation. Some examples of readable types include `simple-arrays`, `lists`, and `structs`. Most readable types are trivially transmissible since from one evaluator we simply pass an object's print representation and at the other evaluator we reconstitute a copy of the object using the built-in `read` function. We also support transmission of some non-readable Common Lisp types like `functions` and `hash tables`. For a more complex type, like object classes, users would need to define their own *marshall* function, which traverses an object's abstract representation and creates a transmissible version, and *unmarshall* function, which reverses that mapping.

As an optimization, we plan to add to our current implementation support for both partial transmission of large objects and transmission of partially evaluated objects. For large readable objects, such as a complex network of `structs`, we would not copy and transmit the entire object but just its root and its descendants up to n-levels deep, where n is user-definable. Hence transmitted objects might include *remote references*, i.e., names of objects that reside remotely. Currently our support for partial transmission of large objects is limited to only immutable `structs`. Finally, for conceptually infinite objects akin to *streams* in Scheme [1], we need transmit only a partial evaluation of their values.

4. Extended Client/Server Model

The client/server model is a common paradigm for distributed computing, especially in systems based on remote procedure call. By introducing remote evaluation into Avalon/Common Lisp we can extend this model in useful and powerful ways. In this section, we explore these extensions, by presenting several models of how distributed programs may be structured in Avalon/Common Lisp .

In the traditional client/server model, the RPC interface serves two purposes. It defines both the calling interface between a client and server and the boundary along which a computation is distributed. The caller (client) is also the initiator process of some computation; the callee (server) is also some initiated process executing on behalf of the caller.

Avalon/Common Lisp separates these two functions. We use the terms *client* and *server* to distinguish between the caller and callee. This client/server distinction defines the interface between the facilities provided by the server programmer, and those provided by the client programmer, just as is true for interfaces in non-distributed programs. We use the terms *local* and *remote* to distinguish between the initiator and initiated processes, i.e., evaluators, of a distributed computation. The local/remote distinction serves to define the boundary along which a computation is distributed. A computation is local if it is performed at the evaluator initiating the computation, while it is remote if it occurs at some evaluator different from the initiator. Remote evaluation is the mechanism by which Avalon/Common Lisp expresses this change in computational locus.

In what follows, we suggest alternative ways to organize the remote and local aspects of client and server interfaces, ranging from traditional RPC to a scheme where both the client and the server do computation both remotely and locally.

As a motivating example, we consider a simple distributed database of bibliography entries such as that used for Scribe or LaTex .bib files. We assume that the user of the database is computing on some local site, e.g., a personal workstation, while the database itself resides on

some remote site. The database interface consists of set operations like `intersection` and `union`; a `matches` function that takes as input a query and returns a set of matching bibliography entries; and a `print-bib-entrys` function that takes as input a set of bibliography entries and returns its print representation. Thus, a typical bibliography database user might write:

```
(print-bib-entrys
  (union
    (matches author-named-Edsger)
    (matches author-named-Butler)))
```

to print all the database entries authored by people named Edsger or Butler.

The Traditional Client/Server Model

To get the effects of RPC as used in the traditional client/server model in Avalon/Common Lisp, we simply put a `remote` around the outermost function call. If the database, the set operations, and the printing and matching functions all reside remotely, then the following code fragment shows how our original single-site query would be expressed:

```
(remote
  (print-bib-entrys
    (union
      (matches author-named-Edsger)
      (matches author-named-Butler))))
```

The Extensible Server Model

In Avalon/Common Lisp, a client can extend a server's interface by transmitting function definitions to the server and can then execute them remotely. In our example, the client first uses a remote `defun` to define a more complicated match function:

```
(remote
  (defun match-Edsgers-or-Butlers ()
    (union
      (matches author-named-Edsger)
      (matches author-named-Butler))))
```

The client executes the following code locally, which evaluates the newly defined function remotely:

```
(remote
  (print-bib-entrys (match-Edsgers-or-Butlers)))
```

A client would normally make multiple remote definitions at one time, perhaps as part of its initialization code. There are several advantages to providing extensible servers. The client programmer gains flexibility by tailoring the server interface to the needs of his or her application. Concrete examples of software with extensible interfaces are Emacs and Postscript [26]. The programmer also can greatly enhance the application's performance by allowing a complex computation to take place near the resource it is manipulating. For example, NeWS [13], an extensible windowing system, can support the smooth rubber-banding of spline curves, while X [20], which essentially uses the standard RPC paradigm, has difficulty smoothly rubber-banding even straight lines.

The Hidden Distribution Model

By permitting some or all server code to run locally, that is, at the local evaluator, Avalon/Common Lisp allows clients to be completely unaware of the distributed nature of a computation. Server writers are free to hide some or all of the distributed aspects of the program from a client. In the most extreme case, the client may never even know that it is using a distributed service.

In the hidden distribution model, our example looks as follows. On the local side, the server writer makes the following definitions (we define macros instead of functions to suppress one level of evaluation):

```
(defmacro matches (query)
  `(remote (matches ,query)))

(defmacro union (setA setB)
  `(remote (union ,setA ,setB)))

(defmacro print-bib-entrys (db)
  `(remote (print-bib-entrys ,db)))
```

The client code is the same as for the non-distributed case:

```
(print-bib-entrys
 (union
  (matches author-named-Edsger)
  (matches author-named-Butler)))
```

Here, when the client calls the three functions provided in the local side of the server code, the server makes the explicit remote calls to the remote side of the server code.

The Full Model

The full model allows both the client and the server to compute both at the local and the remote evaluators. Figure 3 depicts this situation where again, the dotted lines indicate possible call paths. Support for this generality is useful if we want both the ability to

perform complex client computations at the remote site, and to allow the server to hide key aspects of the distributed computation, such as caching.

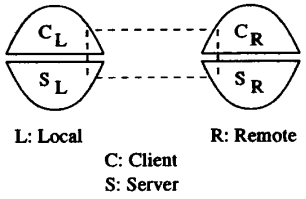

Figure 3: The Full Extended Client/Server Model of Distributed Evaluators

To illustrate both of these capabilities, suppose the server writer implements the `matches` and `union` functions to manipulate the database entries using some compact, but incomplete representation of each entry, while `print-bib-entrys` must have the entire entry before printing it. Caching the complete entries at the local site prevents them from being repeatedly shipped from evaluator to evaluator (i.e., remote to local) while hiding this caching in the server interface allows the client to ignore the complications introduced by the cache.

For our example, on the local side, the server writer makes the following definitions:

```
(defun print-bib-entrys (s)
  (set-map #'print-db-entry s))
(defun print-db-entry (set-entry)
  (unless (value-cached-p set-entry)
    (add-to-cache (get-remote-object set-entry)))
  (print-entry (cached-value set-entry)))
```

The client makes the following remote definition (as before in the Extensible Server Model):

```
(remote
  (defun match-Edsgers-or-Butlers ()
    (union
      (matches author-named-Edsger)
      (matches author-named-Butler))))
```

The client executes it:

```
(print-bib-entrys (remote
          (match-Edsgers-or-Butlers)))
```

The reader should compare the above call to that used in the Extensible Server Model.

The above example illustrates what would commonly be done for querying a large database. In general, application programmers need the ability to write *split queries*, where part of the query is performed remotely through a server interface, and part performed locally through client code. A typical query might be split into a *search predicate* executed remotely and a *filter predicate* executed locally. For example, the search predicate might return a stream of bibliography entries to the client who might then further filter out every fifth entry.

5. Persistence

In this section we show how Avalon/Common Lisp supports failure atomicity through the `with-transaction` construct and persistence through declarations of persistent bindings. We first illustrate these features by showing the relevant pieces of the package for the bibliography database server.

5.1. Example Uses of Extensions

Here we make the database's binding persistent and initialize it:

```
(defpersistent $bib-database$
               (make-persistent (empty-set)))
```

By convention, we use the "$" characters to distinguish those symbols used for persistent bindings from those used for volatile ones. `Make-persistent` creates a recoverable object; `defpersistent` defines `$bib-database$` as a special symbol whose binding is recoverable, and creates a binding between `$bib-database$` and the recoverable empty set.

We use transactions for standard database operations such as adding, modifying, and deleting entries. Consider the function for adding a bibliography entry:

```
(defun add-bib-entry (entry)
  (with-transaction
   (if (valid-bib-entry-p entry)
       (adjoin $bib-database$
               (make-persistent entry))
       (abort-transaction 'invalid-bib-entry))))
```

If the entry is valid, i.e., well-formed and not already in the database, then we make the volatile value of the `entry` argument persistent and add it to the database. Otherwise, we abort the transaction signalling

the abort condition `invalid-bib-entry`. Since the update is done within a transaction, if a crash occurs during the update, the state of the bibliography database will be as if the update never occurred; Camelot's recovery algorithm will guarantee the database is restored to a previously saved consistent state.

The counterpart to `make-persistent` is `make-volatile`. Since an evaluator communicates with a recoverable store, retrieving a persistent binding from it gives us a handle on a recoverable object. Upon retrieval, we are free to continue to use the object as a recoverable object until we need to either call a standard Common Lisp function or transmit the object back to the local evaluator. Thus as a server writer, we have some latitude as to when to make the `make-volatile` call. For example, both `print-bib-entrys` below have the same eventual effect:

```
(defun print-bib-entrys ()
  (set-mapc #'(lambda (set-entry)
                (print-bib-entry
                  (make-volatile set-entry)))
                  $bib-database$))

(defun print-bib-entrys ()
  (set-mapc #'print-bib-entry
                (make-volatile $bib-database$)))
```

In the first version, `set-mapc` operates on a persistent set (and uses `rec-car`, `rec-cdr`, etc. to traverse the `$bib-database$`[3]). In the second, `set-mapc` operates on a volatile object. `Make-persistent` and `make-volatile` are each idempotent and are inverses of each other.

Avalon/Common Lisp currently supports recoverable versions of a large subset of Common Lisp's built-in types, e.g., `fixnum`, `list`, `simple-string`, `simple-vector`, as well as any type constructed using `struct`'s.

5.2. New Macros and Functions

Here is the programmer's interface to the new macros and functions:

`defpersistent` *variable* [*initial-value*] [*Macro*]

This form is similar to the `defvar` form, except that any binding to *variable* is recoverable, i.e., survives crashes and supports failure atomicity. If given, *initial-value* is assigned to *variable*, as long as *variable* has not previously been bound. *Initial-value* must evaluate to a recoverable object and is only evaluated if it is used to initialize the binding.

All subsequent `setq` operations to *variable* will change the binding atomically; `setq` operations to persistent variables can be aborted if evaluated within a transaction.

`make-persistent` *object* [*Function*]
`make-volatile` *object* [*Function*]

These functions create a persistent (volatile) representation of *object*. If *object* is already persistent (volatile), it is returned as the result.

`with-transaction` *body* [*Macro*]
`with-top-level-transaction` *body* [*Macro*]

Both forms initiate a new transaction and evaluate *body*. `With-transaction`, if evaluated dynamically within another transaction, will begin a nested transaction; otherwise it starts a top-level transaction. `With-top-level-transaction` always initiates a new top-level transaction. Both forms return a mutiple value consisting of a status signifying whether or not the transaction committed, and the result of the last expression in *body*.

Normal evaluation of either form results in a committed transaction. Exceptional exits from the *body* (via `catch/throw` and local exits) result in the transaction aborting. Transactions can also be explicitly aborted via use of `abort-transaction`.

`abort-transaction` *retval* `&optional top-level`[*Macro*]

This form aborts the currently executing transaction. If the optional argument is nil (the default), the innermost dynamically nested transaction is aborted and the value of *retval* is returned as the status in the multiple-value result of `with-transaction`. Otherwise, the current (dynamically scoped) top-level transaction is aborted.

6. Related Work and Discussion

Our work on remote evaluation is closest in spirit to Stamos's Ph.D. work [23] for which he designed extensions to the programming language CLU to support remote evaluation in the context of atomic transactions. Since the target languages differ, so do our concrete designs. We designed and packaged our language extensions in a way that avoids modifying the compiler and instead exploits the interpretative programming style of Common Lisp. Since CLU is a compile-time (strongly) typed language, Stamos defines static checks

[3] Avalon/Common Lisp supports "recoverable" versions of some standard Lisp functions like `car`, `cdr`, `eq`, `eql`, etc. They operate on objects retrieved from recoverable store, rather than normal non-recoverable Lisp objects.

that must be performed to ensure a remote evaluation request is valid. Client extensions to servers and code arguments further complicate both these checks and the compiler's subsequent encoding of a remote evaluation request. We avoid some of these difficulties since our new **evaluator** data type gives us not only a run-time boundary (each is a process), but a compile-time boundary (each defines a global namespace for a set of packages).

Our extensions to the client/server model are similar to that supported by Falcone's Heterogeneous Distributed System architecture, prototyped at DEC [10]. Falcone focuses on support at the operating-system level, rather than at the programming-language level, though he does provide a small Lisp-like language interface to the system facilities. By our extending Common Lisp rather than defining a new language, we have the advantage of completely integrating our extensions with an existing, familiar, and widely-available programming language. Also, Falcone handles only primitive data types such as lists and byte vectors, and does not address persistent and recoverable storage of data.

Avalon/Common Lisp is distinct from other distributed programming languages such as CSP [15], SR [2], Linda [12], Nil [24], and Ada [19], since we have direct support not only for remote evaluation, but for transactions, and in particular the following features: commit and abort processing, crash recovery, atomic objects, and management of persistent data.

Even though Avalon/Common Lisp lacks light-weight processes, there are some similarities between it and other "concurrent Lisp" efforts such as Qlisp [11] and Multilisp [11], both of which support concurrent computation using light-weight, shared-memory processes. Qlisp's *qlambda* construct creates a closure and a process that will be used to evaluate any future application of the closure. Like qlambda instances, Avalon/Common Lisp's evaluators are each identified with a separate process. The lexical environment inherited in a `remote` call could be passed to a Qlisp process as an argument, and the evaluator-specific global environment could be simulated using the environment of the qlambda's closure. Avalon/Common Lisp's `remote` construct is also similar to Multilisp's *future* construct. Both allow the programmer to dispatch arbitrary forms to another process for evaluation. A key difference is that Multilisp's processes, being light-weight, are created on-demand, while the evaluators in Avalon/Common Lisp are heavy-weight, and, therefore, long-lived.

Transactions themselves have been a primary focus in both distributed and centralized data bases ([3], [8], [14], [9]). Several research projects have chosen transactions as the foundation for constructing reliable general-purpose distributed programs, including Argus [17], Arjuna [7], Clouds [18], TABS [22], and Camelot [21]. Of these projects, however, only Argus and Arjuna have addressed the linguistic aspects of the problem. Argus extends CLU and Arjuna extends C++. None of these projects have direct support for remote evaluation or our extended client/server model.

Avalon/C++ and Avalon/Common Lisp differ in significant ways even though they address the same application domain, reliable distributed computing, and are motivated by the need to provide programming level support for transactions. Avalon/C++'s primary design focus was on user-defined atomic data types, in particular, support for *hybrid atomicity*. Programmers can define (hybrid atomic) objects that provide higher degrees of concurrency than that provided by using standard two-phase read/write locks, such as that used for Avalon/Common Lisp. In contrast, Avalon/Common Lisp's primary design focus is on remote evaluation and support for a client/server model more general and flexible than the traditional one such as that used for Avalon/C++. Thus, Avalon/Common Lisp relies on well-known techniques for dealing with serializability (read/write locks) and persistence (write-ahead logging, recoverable virtual memory), but introduces a new model for distributed computing.

Currently, all Avalon software runs on IBM RT's in the Mach and Camelot environments. Avalon/C++ runs on Sun's and Vaxes as well. Avalon/C++ has been operational for a year and we are not doing any further design or implementation work with it.

Avalon/Common Lisp is nearly complete as of this writing. All Avalon/Common Lisp code presented in this paper runs. Besides the bibliography database, other Avalon/Common Lisp examples include a simple array server and a factory-parts database. Further details on the Avalon/Common Lisp programmer's interface are in [4].

In summary, Avalon is a set of linguistic constructs that extend the capability of existing programming languages by directly supporting transactions. For each of our target languages, C++ and Common Lisp, we designed our extensions to be unintrusive and modular. For example, a Common Lisp programmer can load one set of packages if support for only remote evaluation is desired, a different set if support for only recoverable store is desired, or both sets if both features are desired. These language extensions relieve users from the burden of doing low-level system activities such as locking and managing stable storage, and instead allow them to concentrate on the logic required of their application. At the same time, however, they are given enough flexibility to exploit the semantics of their applications to increase their programs' reliability and efficiency.

Acknowledgments

We thank Gene Rollins who has helped motivate some of the design of Avalon/Common Lisp through his willingness to be our first user. He and Karen Kietzke also gave useful comments on earlier drafts of this paper.

We also thank Dave McDonald for providing us with a barebones Common Lisp interface to Camelot's recoverable storage manager, and Alfred Spector and the rest of the former Camelot crew for providing us with basic run-time support.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract No. F33615-87-C-1499. Additional support was provided in part for J.M. Wing by the National Science Foundation under grant CCR-8620027 and for S.M. Clamen by the Office for Naval Research under grant ONR-N00014-88-12-0641.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the National Science Foundation, the Office for Naval Research or the U.S. Government.

References

[1] H. Abelson and G.J. Sussman with J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[2] G.R. Andrews. Synchronizing resources. *ACM Transactions on Programming Languages and Systems*, 3(4):405–430, October 1981.

[3] P.A. Bernstein and N. Goodman. A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Computing Surveys*, 13(2):185–222, June 1981.

[4] S.M. Clamen, L.D. Leibengood, S.M. Nettles, and J.M. Wing. The avalon/common lisp programmer's guide. 1989. Avalon Design Note 14.

[5] Eric C. Cooper and Richard P. Draves. *C Threads*. Technical Report CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, June 1988.

[6] D.L. Detlefs, M.P. Herlihy, and J.M. Wing. Inheritance of synchronization and recovery properties in avalonc++. *IEEE Computer*, 57–69, December 1988.

[7] G. Dixon and S.K. Shrivastava. Exploiting type inheritance facilities to implement recoverability in object based systems. In *Proceedings of the Sixth IEEE Symposium on Reliability in Distributed Software and Database Systems*, 1987.

[8] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notion of consistency and predicate locks in a database system. *Communications ACM*, 19(11):624–633, November 1976.

[9] B.G. Lindsay et al. *Notes on Distributed Databases*. Technical Report RJ2571, IBM San Jose Research Laboratory, July 1979.

[10] J.R. Falcone. A programmable interface language for heterogeneous distributed systems. *ACM Transactions on Computer Systems*, 5(4), November 1987.

[11] R. P. Gabriel and J. McCarthy. Queue-based multi-processing lisp. In *ACM Symposium on Lisp and Functional Programming*, pages 25–44, August 1984.

[12] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[13] J. Gosling, D.S.H. Rosenthal, and M. J. Arden. *The NeWS Book: An Introduction to the Networked Extensible Window System*. Springer-Verlag, 1989.

[14] J.N. Gray. *Notes on Database Operating Systems*, pages 393–481. Springer-Verlag, Berlin, 1978.

[15] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[16] G. Steele Jr. *Common LISP*. Digital Press, 1984.

[17] B. Liskov and R. Scheifler. Guardians and actions: linguistic support for robust, distributed programs. *Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[18] M.S. McKendry. Clouds: a fault-tolerant distributed operating system. *IEEE Tech. Com. Distributed Processing Newsletter*, 2(6), June 1984.

[19] Dept. of Defense. Reference manual for the ada programming language. 1983. ANSI/MIL-STD-1815A-1983.

[20] R.W. Scheifler and J. Gettys. The x window system. *ACM Transactions on Graphics*, 5(2), April 1986.

[21] A.Z. Spector, J.J. Bloch, D.S. Daniels, R.P. Draves, D. Duchamp, J.L. Eppinger, S.G. Menees, and D.S. Thompson. The camelot project. *Database Engineering*, 9(4), December 1986. Also available as Technical Report CMU-CS-86-166, Carnegie Mellon University, November 1986.

[22] A.Z. Spector, J. Butcher, D.S. Daniels, J.L. Eppinger D.J. Duchamp, C.E. Fineman, A. Heddaya, and P.M. Schwarz. Support for distributed transactions in the tabs prototype. *IEEE Transactions on Software Engineering*, 11(6):520–530, June 1985.

[23] J.W. Stamos. *Remote Evaluation*. Technical Report 354, MIT Labxooratory for Computer Science, January 1986.

[24] R.E. Strom and S. Yemini. Nil: an integrated language and system for distributed programming. In *SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, June 1983. Also an IBM research report (RC 9499 (44100)) from April 1983.

[25] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.

[26] Adobe Systems. *Postscript Language Reference Manual*. Addison-Wesley, 1985.