Support for Pointer Semantics in a Generative Communication Framework

Richard L. Neil

Dissertation submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
In
Computer Science

Keywords: Pointer Semantics, Generative Communication, Asynchronous Communication, Concurrent Access

Support for Pointer Semantics in a Generative Communication Framework

Richard L. Neil

Abstract

The Generative Communication (GC) paradigm was introduced in the early 1980s in the form of the Linda coordination and communication language (Gelernter & Bernstein, 1982). The goal of the language was to provide an elegant, intuitive alternative to the state-of-the-art in parallel and distributed processing languages at that time, namely remote procedure calls; message passing systems; and shared memory systems (Gelernter, 1985). Despite the longevity of the Linda coordination language and the GC paradigm, in general, the literature is devoid of meaningful research into the construction and manipulation of dynamic, pointer-based data structures within the context of the GC paradigm. This document motivates the need for such data structures in GC. In addition, this document explores the reasons for the lack literature regarding dynamic pointer-based data structures in GC. Finally, we develop solutions to address the lack of support, within the GC paradigm, for such data structures.

Table of Contents

List of Figures

List of Tables

Chapter 1 – Introduction

Concurrent programming languages may be viewed as a composite of two elements. One element is concerned with computation while the other is concerned with communication and coordination (Carriero & Gelernter, 1992). The Generative Communication (GC) paradigm was introduced in the early 1980s in the form of the Linda coordination and communication language (Gelernter & Bernstein, 1982). The goal of the language was to provide an elegant, intuitive alternative to the state-of-the-art in parallel and distributed processing languages at that time, namely remote procedure calls; message passing systems; and shared memory systems (Gelernter, 1985).

Linda, when incorporated into a sequential programming language (hereafter referred to as the host language), results in a powerful concurrent programming language. Linda derives its simplicity and elegance from the fact that it is not entangled with the computational component of the host language, as opposed to the so-called "Big Three" among its contemporaries – object-oriented programming, logic programming, and functional programming (Carriero & Gelernter, 1989a). In contrast to those languages, in which the lines between the two elements are blurred, all that is required to begin expressing concurrency using Linda is a fundamental understanding of six operations added to the host language, and an understanding of the semantics of an abstract, shared data space called tuplespace (TS).

In addition to the relative ease with which concurrency can be expressed using GC, GC offers much more flexibility than other models of concurrency. Whereas other paradigms require synchronization between concurrent processes in terms of time, in terms of space, or in terms of both time and space, GC remains decoupled across both dimensions. Consider Communicating Sequential Processes (CSP) as an example (Hoare, 1978). In CSP, for two processes to communicate, the sender must first explicitly identify the receiving process and wait for a response. The receiver must explicitly identify the sending process and wait for a response. When both processes are ready, only then can the communication take place. It is clear that CSP is tightly coupled, both spatially (both parties to the communication must mutually identify each other), and temporally (both parties must be ready to take part in the communication at the same time).

To understand the flexibility of GC, consider a similar scenario involving Linda. As in the CSP example, one process needs to send a message to another process. In this case, however, the sender simply deposits the message into TS without regard to the identity of the receiver and without regard to when the receiver process will retrieve the message. When the receiver process is ready to retrieve the message, it merely removes the message from TS and proceeds.

This is not to imply, however, that handshaking is impossible between two processes communicating via Linda (Gelernter, 1985). The semantics of the Linda operations are covered in detail in Chapter 2. However, by way of introduction consider that there are two choices of operations to retrieve data from TS. One operation will, if no matching tuple exists in TS, block until a matching tuple is placed into TS. The other operation will not. The latter operation fails and returns control to the calling process. The blocking semantics of the former operation allows Linda to simulate the spatial and temporal coupling inherent to CSP, as illustrated in the following scenario.

> A reader process makes a request for a tuple from a certain sender. The tuple does not yet exist in TS, so the reader process blocks. The sender process places a tuple in TS identifying itself as the sender. The sender process then attempts to remove a tuple from TS indicating that the specified reader process has received the message. No such tuple exists, so the sender process blocks. In the mean time, the reader process has unblocked and removed the tuple containing the message. The receiver process then generates an acknowledgement tuple and places it in TS. This causes the sender process to unblock and retrieve the acknowledgement tuple.

So revolutionary was Linda that it spawned a plethora of spin-off implementations such as Glenda (Seyfarth, Bickham, & Arumugham, 1993); p4-Linda (Butler, Leveton, & Lusk, 1993); Ada-Linda (Kermarrec & Pautet, 1994); Bauhaus (Carriero, Gelernter, & Hupfer, 1997); BaLinda (Yuen & Feng, 1997 & 1998); KLAIM (De Nicola, Ferrari, & Pugliese, 1998); X-KLAIM (Bettini, De Nicola, Pugliese, & Ferrari, 1998); and LIME (Picco, Murphy, & Roman, 1999; Murphy, Picco, & Roman, 2001). Several object-oriented derivatives of Linda have also been produced. For example, ObjectSpace (Polze, 1993) and BaLinda C++ (Wang & Feng, 1999). Finally, perhaps the most notable of Linda's object-oriented cousins are JavaSpaces (Sun Microsystems, 2002) and TSpaces (Wyckoff, 1998; Lehman, McLaughry, & Wyckoff, 1999).

Despite C. A. R. Hoare's (1975) admonition that the inclusion of reference type variables in high-level programming languages was a "retrograde step", the utility of dynamic pointer-based data structures (hereafter referred to as, Linked Data Structures) is undeniable. The typical representation of LDS uses a record structure in which at least one field is a recursively-defined pointer to instances of the same record structure type (see Figure 1.1). Instances of the record type – referred to as the nodes of the LDS – are chained together by setting the value associated with the pointer field of one node to refer to the memory location at which the next node is stored.

```
struct node {
    char[10] data;
    struct node *next;
}
```

Figure 1.1 - Definition of a simple LDS node structure in

Computationally intensive applications such as image processing systems, geographical information systems, and data mining systems make extensive use of pointers to dynamically construct LDS (Chang, Sussman, & Saltz, 1995). Among the benefits of LDS is the ability to expand and shrink dynamically according to the need of the application. Additionally, LDS can be distributed within the storage medium (i.e., the programmer does not have to reserve a contiguous block of storage to contain an LDS).

Finally, LDS can be reordered logically by manipulating pointers without moving large amounts of data (vis-à-vis maintaining order in an array, for example).

Given the ease and flexibility with which concurrency can be expressed in GC, and the ubiquity and utility of LDS, one might logically conclude that the combination of GC and LDS would offer the best of both worlds. However, in their discussion of distributed data structures in Linda, Carriero, Gelernter, and Leichter (1986) relegate LDS to the status of a parenthetical remark. They discuss, in great detail, unordered structures such as bags; and ordered structures such as arrays, matrices, and broadcast streams. The only mention of LDS is in reference to a programming language called Symmetric Lisp in which lists are built by replacing physical addresses with logical names. The same treatment of LDS appears in a subsequent paper by Carriero and Gelernter (1989b). They also reveal that LDS in TS received only cursory treatment in their programming experiments. In fact, the literature seems utterly devoid of any meaningful coverage of the subject. Section 1.1 examines the possible reasons for LDS research in GC remaining on the periphery.

1.1 – Problems Creating Linked Data Structures in TS

We begin exploring the nature and etiology of the issues, with respect to the construction of LDS in GC, by examining LDS within the context of other models of concurrency. True concurrency can only be achieved through the use of Single Instruction, Multiple Data-stream (SIMD) architectures or Multiple Instruction, Multiple Data-stream architectures (MIMD). We will ignore SIMD architectures because they embody a paradigm that is not relevant to the current discussion – one instruction executed in lockstep by some number of processors, each with its own dataset on which to operate (Tanenbaum, 1999). We focus, instead, on MIMD architectures. The broader class of MIMD processor organizations is further subdivided into tightly coupled architectures and loosely coupled architectures (Stallings, 2003). Tightly coupled architectures share a common address space (Stallings, 2003). Examples of tightly coupled architectures are the Symmetric Multiprocessor (SMP) machines and the Nonuinform Memory Access (NUMA) machines. Loosely coupled architectures, on the other hand, are characterized by distributed memories, each with its own address space (Stallings, 2003). These architectures are referred to as clusters (Stallings, 2003) or multicomputer architectures (Tanenbaum, 1999). An example of a loosely coupled machine is the Intel Hypercube (Singhal & Shivaratri, 1994).

Consider, now, the task of constructing a linked list to be concurrently manipulated by processes running on a tightly coupled machine. Clearly there is a need for synchronization, between processes operating on the list, to prevent the structure and contents of the list from becoming corrupted. However, the shared memory and single address space provide the same "raw materials", for the construction of an LDS, that are available to a single processor with exclusive access to its address space. Therefore, tightly coupled MIMD architectures provide us with no information as to the source of our issues with LDS in GC, nor do they provide any information with respect to the solutions to those issues.

Perhaps a key to understanding the problem can be found within the realm of loosely coupled MIMD architectures. These architectures have no common address space through which processes can share data. They must, instead, rely upon message passing or remote procedure calls (Singhal & Shivaratri, 1994). The implications of this are clear – any data structures on which processes operate under this environment must reside somewhere in the unique address space of one or more of the processors. This means that LDS can still be constructed using the traditional method of stringing nodes together using pointers to the memory locations where they are stored. Obviously the access method to the LDS must change under this paradigm. Instead of dereferencing pointers to access the data of a node, a message must be sent or a remote procedure must be called. However, the fundamental construction of the LDS remains the same as it is under tightly coupled MIMD architectures.

It is widely recognized that loosely coupled MIMD systems are difficult to program (Bal & Tanenbaum, 1988; Drucker & Frank, 1996; Baratloo, Dasgupta, & Kedem, 1995; Keleher, Cox, Dwarkadas, & Zwaenepoel, 1994; and Nikhil, 1994). In an attempt to mitigate the difficulties inherent in programming a loosely coupled MIMD system, several systems exist to provide the programmer with the illusion of a single global address space. The dominant paradigms within this genre generally fall into two categories. The first is data replication – copies of the data in the memories of each processor are replicated on every other processor as needed (e.g., Keleher, Cox, Dwarkadas & Zwaenepoel, 1994). The second technique involves global pointers. In this model, processors essentially allow other processors access to their address spaces via global pointers. The global pointers consist of processor identification and a pointer to a location within the address space of that processor (e.g., Barataloo, Dasgupta, & Kedem, 1995; Chang, Sussman, & Saltz, 1995; and Nikhil, 1994). While these transparent distributed shared memory (DSM) models might make programming easier, they still involve the construction of an LDS within the address space of one processor or another. The access semantics have been made more natural, but the underlying "bricks and mortar", with which the LDS is constructed, remains unchanged. This is not the case with the GC paradigm.

In theory, any data for which there exists a representation in the host language can be represented within a tuple and passed to TS. In practice, however, there are several significant issues that must be considered when enhancing a sequential programming language with a GC implementation such as Linda (Ciancarini, 1994). One such consideration in combining Linda with the C programming language is the loss of the ability to pass LDS to TS. As described above, in the typical representation of an LDS, nodes are chained together by setting the value associated with the pointer field of one node to refer to the memory location at which the next node is stored. Therein lies the root of the problem with respect to constructing LDS in TS: the GC paradigm makes no provision for pointer semantics.

This fact certainly makes sense, given that traditional pointer semantics utilize a one-to-one mapping between a memory address and the location it identifies. Removed from the context of the local address space, a pointer is meaningless (Ciancarini, 1994). In fact, removed from the context of a particular moment in time during the execution of a particular program, a pointer is meaningless (Hoare, 1975). In GC, there are no tuple addresses because there are no addressable locations (Carriero & Gelernter, 1988). Instead, tuplespace is viewed, in the abstract, as bag or multi-set of tuples (Campbell, Osborne, & Wood, 1997). Tuples are placed therein without regard to how they are stored. Clearly, tuples are stored in physical memory. The specific mechanism by which this is accomplished, however, is irrelevant to GC semantics and is left to the discretion of the implementer (Fleckenstein & Hemmendinger, 1989).

Tuple identification in TS is based, not upon the location of the tuple, but rather upon the contents of the tuple. To retrieve a tuple from TS, the programmer uses a template consisting of one or more typed fields referred to as parameters. Each parameter is either an actual parameter or a formal parameter (Narem, 1989). In the case of the former, the parameter is associated with the representation of a value. In the case of the latter, a representation of a value will be associated with the parameter when a tuple, matching the template, is found in TS. Matching is based upon the parameter signature of the tuple. The parameter signature consists of the relative field positions within the template, the data type associated with each field, and for actual parameters, their associated values. From the pool of tuples in TS, that have the same parameter signature as the template, one is non-deterministically selected and returned to the calling process.

The GC paradigm has made a recent resurgence in the form of Sun Microsystems' JavaSpaces (Sun Microsystems, 2002) and IBM's TSpaces (Wyckoff, 1998; Lehman, McLaughry, & Wyckoff, 1999; and Lehman, Cozzi, Yuhong, Gottschalk, Vasudeval, Landis, et al., 2001). Java objects can refer to other objects within the address space of a particular Java Virtual Machine (JVM). A reference to a class instance in Sun's JVM is a pointer to an object handle. The object handle contains a pair of pointers. The first pointer points to a table containing the object's methods and a pointer to the class object upon which the current instance is based. The second pointer, in the handle, points to the memory allocated from the heap for the object's data (Lindholm & Yellin, 1996). In other words, Java objects are linked together with pointers. These very same objects can then be stored in JavaSpace or a TSpace. Does this mean that the issue of pointers in GC has been solved?

The answer to that question is no. While it is true that objects use pointers to refer to other objects within the local address space of the JVM, those pointers do not transfer to JavaSpace or TSpace. Both systems rely upon the facilities of the *java.io.Serializable* interface to marshal objects on their way to JavaSpace or TSpace, and to un-marshal them when they are retrieved from JavaSpace or TSpace. As objects are marshaled by *java.io.Serializable,* all of the objects, to which they refer, are also marshaled and packaged along with them. In addition to a representation of the objects' data and methods, the serialized objects are also packaged with information that will allow the objects' states to be instantiated in the local address space of (possibly) another JVM when the objects are removed from JavaSpace or TSpace.

On the surface, it sounds as if the ability to reconstruct the relationship between objects, and the objects they reference, would embody the solution to the "pointers in TS problem". However, there are two aspects of this technology that make Java's serialization mechanism untenable as a solution to this problem. First, consider the case where the root or head of the LDS is sent to one of the systems in question. All of the objects to which it refers will be marshaled and sent with it. In turn, all of the objects to which they refer will receive similar treatment, as do all of the objects to which they refer, and so on. The serialized form of the entire LDS is stored as a single unit in the case of both JavaSpaces and TSpaces. Therefore, in the case of an LDS, it would be impossible for more than one process to operate upon different nodes of the LDS simultaneously.

To understand the second problematic aspect of these systems, consider the scenario in which the programmer manually deconstructs the LDS (see Figure 1.2) and stores each node individually to JavaSpace or TSpace starting with the root, proceeding to the next node, etc., until the end of the LDS has been reached.

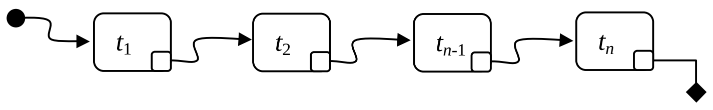


Figure 1.2 – Example singly-linked list.

As illustrated in Figure 1.3, the side-effect of the *java.io.Serializable* functionality, used in this context, will lead to 1 copy of the first node being stored, 2 copies of the second node, *n-1* copies of node *n-1*, and finally, *n* copies of the *n*th node.

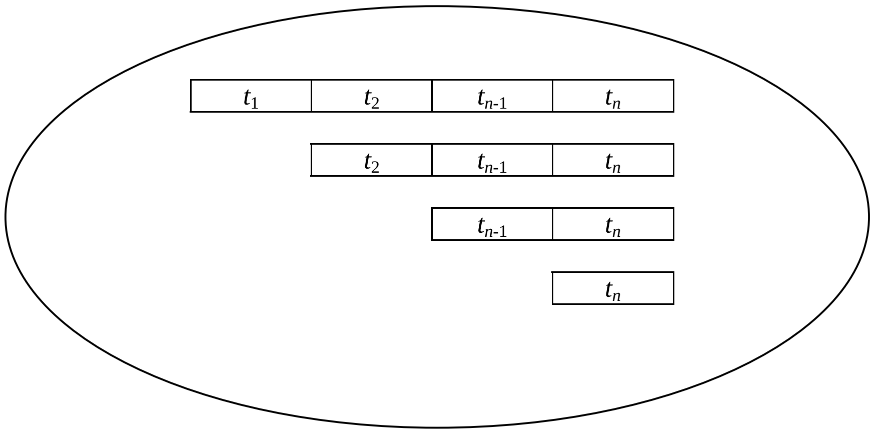


Figure 1.3 – Example singly-linked list stored in JavaSpaces or TSpaces

The results are even more dramatic when the nodes of the LDS are retrieved from JavaSpace or TSpace. Consider a simple situation in which two objects in the local address space of a JVM refer to one other object as illustrated by Figure 1.4.

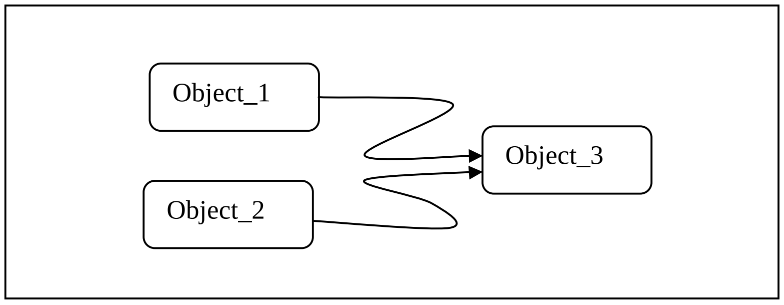


Figure 1.4 – Java objects in RAM at time Θ_0.

The two objects, Object_1 and Object_2, are stored in JavaSpace at time Θ_1, as illustrated by Figure 1.5.

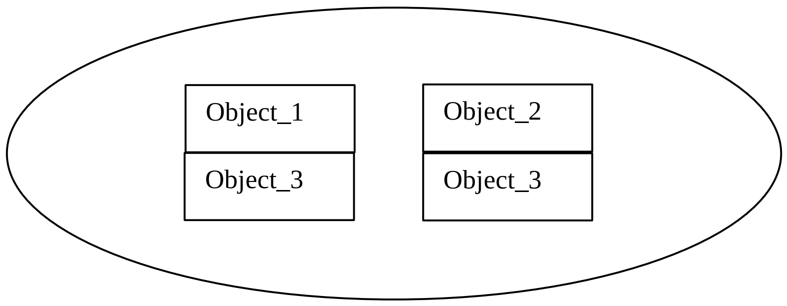


Figure 1.5 – Java objects in JavaSpace at time Θ_1.

As Figure 1.5 shows, the objects are stored in JavaSpace with two separate copies of Object_3. The next figure (Figure 1.6) shows the results when the two objects are retrieved from JavaSpace at time Θ_2.

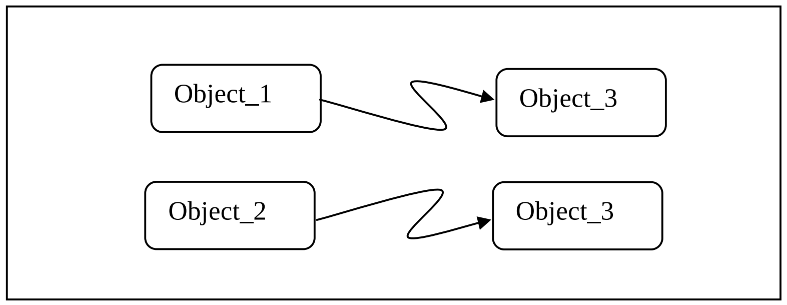


Figure 1.6 – Java objects in RAM at time Θ_2.

As we see from this scenario, two objects that previously referenced one copy of a third object now both refer to a unique copy of that object. The implications of this for a linked list are as follows: One list with *n* nodes becomes *n* lists, each with a number of nodes between 1 and *n*, when it is stored in JavaSpace or TSpace and then retrieved back into the address space of the JVM.

1.2 – Goals of the Current Research

Content-based, associative addressing is the source of the problems involved with constructing LDS in TS and concurrently operating upon them. The present research addresses this shortcoming by incorporating support for pointer semantics and, by extension, support for LDS, into the GC paradigm. Additionally this work seeks to facilitate the practical, safe, and efficient concurrent manipulation of LDS created in TS using the newly incorporated pointer semantics. There are numerous GC implementations we could have selected to exemplify and implement our theoretical solutions to the pointer semantics problem. We chose Linda for two reasons. First, it is the lowest common denominator among all GC implementations. There exists no known GC implementation that does not use Linda as its jumping-off point. Second, Linda remains unencumbered with many of the so-called enhancements to the GC paradigm that have driven its successors (e.g., object-orientation, reactive tuplespace, security considerations, etc.). We selected C-Linda, in particular, because of the ubiquitous nature of the C language within the research community.

1.3 – The Solution Approach

The successful solution adheres, as much as possible, to the following overarching principles we have set forth to constrain the solution space:

- First, and foremost, the pointer semantics, and the operations thereon, must ensure the structural integrity and data integrity of the LDS constructed with them. We will elaborate upon the definitions of the terms structural integrity and data integrity as we discuss the solution and potential threats to both types of integrity. However, loosely defined, data integrity means that the data contained within the nodes of an LDS have not been corrupted as a result of operations on the LDS. Structural integrity refers to ensuring that the connections between nodes of the LDS remain intact following operations on the LDS – that is to say we do not end up with broken links, dangling pointers, etc.
- Second, we wish to remain faithful to both traditional pointer semantics and existing GC semantics.
- Third, we develop our solutions under the assumption that we know nothing about the intent of the programmer. In several respects, this choice has made the task more difficult. However, it also serves to make the solutions more robust because we are not concerned with the consequences of making incorrect assumptions about the intent of the programmer.

- Finally, insofar as we selected C-Linda as the particular vehicle for our research, we endeavor to remain faithful to its particular flavor of GC semantics and its existing syntactical conventions.

It is impossible to optimize two diametrically opposing properties. We have demonstrated that pointer semantics and GC semantics do, in fact, diametrically oppose one another. Nevertheless, we strive to structure the inevitable trade-offs in such a way so as to achieve the maximum possible benefit while giving up as little as possible with respect to the ideals.

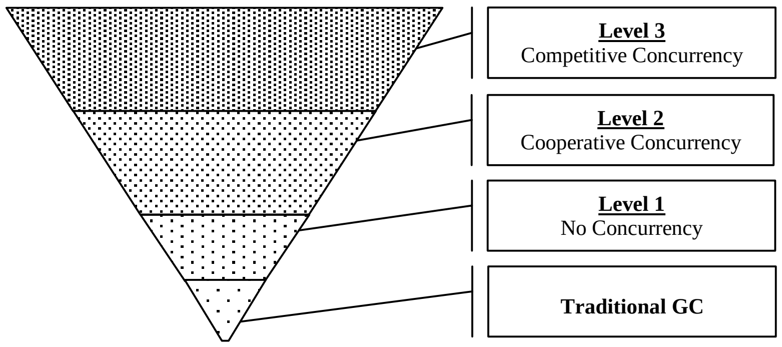


Figure 1.7 – Structure of the solution approach.

As Figure 1.7 illustrates, we attack the problem using a bottom-up strategy. Beginning with traditional C-Linda as our representative of GC, we artificially constrain the problem by eliminating concurrency from consideration, yielding Level 1. At Level 1, we are only concerned with the necessary and sufficient conditions to support pointer semantics. Once the issues at Level 1 have been addressed, we widen the scope of the problem a bit more and introduce cooperative concurrency into the mix – we refer to this as Level 2. Level 2 allows us to identify the issues that arise as a result of concurrency, while still restricting the scope of the problem by eliminating competition. Once we identify and resolve the issues surrounding cooperative concurrent access to LDS constructed with GC pointers, we can widen the problem, yet again, by allowing competitive concurrent access to occur. Level 3 consists of Linda with the necessary and sufficient primitives to allow the construction and competitive concurrent manipulation of LDS in TS. As proof of concept, we have implemented a prototypical C-Linda With

Pointers (CLWP) system that includes support for the functionality of Level 1 through Level 3.

The remainder of this document is organized as follows. Chapter 2 provides a brief GC and C-Linda primer. Following Chapter 2, we discuss, in Chapter 3, the issues involved with developing Level 1 and the resolutions of those issues. Chapter 4 and Chapter 5 treat Level 2 and Level 3 similarly. The architecture of the prototype, and the issues surrounding its implementation, are discussed in Chapter 6. A summary of the current research, an outline of work yet to be completed, a schedule for completing the remaining work, and concluding remarks appear in Chapter 7. Appendices A-B contain example C-Linda code to construct and manipulate a singly-linked list in TS using Level 1 and Level 2 functionality, respectively.

Chapter 2 – Overview of GC and C-Linda

The GC paradigm in general, and C-Linda in particular, consists of two major components. The first component is a shared, abstract data space referred to as tuplespace (TS). The second component of GC is the tuple, which can either be passive or active (Gelernter, 1985). Active tuples contain executable code and are placed into TS using the *eval* operation (Carriero & Gelernter, 1989a). Active tuples and the *eval* operation are not germane to the current research so they are beyond the scope of this document. The passive tuple is an ordered list of one or more typed fields, each being either a formal field or an actual field (Narem, 1989). Six operations are used to interact with TS:

- *out* – places a tuple in TS;
- *in* – removes a tuple from TS and returns the contents to the calling process – block waiting for a matching tuple if none exists;
- *rd* – similar to *in* except the tuple remains in TS.
- *inp* – same as *in* except the operation fails and returns, instead of blocking, if no matching tuple is found.
- *rdp* – same as *rd* except the operation fails and returns, instead of blocking, if no matching tuple is found.
- *eval* – place an active tuple in TS to be evaluated.

When a process needs to pass data to another process it generates a tuple and places the tuple into TS by executing an *out* operation as in the following example (see Figure 2.1).

```
int x = 60;
out(“example1”, 10.5, x);
```

Figure 2.1 – Example *out* operation

To retrieve the tuple from TS, a process simply executes a *rd* operation or an *in* operation. The *rd* operation returns a copy of the original tuple while leaving the original in TS. The *in* operation, on the other hand, removes the tuple from TS and returns it to the calling process. If no matching tuple is found, the *in* and the *rd* operations will block until a matching tuple is added to TS. There also exist non-blocking forms, or predicate forms, of both operations named *inp* and *rdp* respectively. These operations behave exactly as their blocking counterparts with the exception that they do not block if no matching tuple is found. They simply fail and return a code (and control) to the calling process indicating that no match was found (Carriero & Gelernter, 1989a).

```
int y;
double z;
in(“example1”, ?z, ?y);
```

Figure 2.2 – Example *in* operation

When a process executes one of the *rd*, the *in*, the *rdp*, or the *inp* operations, the identification of matching tuples, if any exist, proceeds by comparing parameter signatures of the tuples in TS and the parameter signature of the template specified by the programmer. The template looks just like a tuple with one exception: the parameters of the template can be either formal or actual. An actual parameter contains data. A formal parameter, on the other hand, is a data receptacle with which a value is associated if a matching tuple is found in TS. Parameter signatures are constructed by comparing the data type, the relative position of the parameter within in the tuple or template, and the value associated with the parameter if it is an actual parameter. If the parameter signature of the template matches the parameter signature of one or more tuples in TS, then the appropriate action is taken for the operation in question – including non-deterministically selecting one of the matching tuples, and replacing formal parameters with their corresponding actual values from the selected matching tuple. If no tuple is found in TS with a parameter signature matching that of the specified template, then the blocking operations will block and the predicate forms of the operations will fail and return an appropriate code to the calling procedure. The code segment above (see Figure 2.2) demonstrates the use of the *in* operation. This command retrieves the tuple placed in TS by the code segment in Figure 2.1. Following the command, the variables identified by `z` and `y` would be associated with representations of the values `10.5` and `60`, respectively.

Chapter 3 – Pointer Support for Generative Communication: Level 1

This chapter examines the construction and manipulation of LDS in TS within the simplest computational environment possible: No concurrent access to the LDS in TS is permitted.

This chapter addresses the following issues:

- How do we achieve pointer semantics in a shared data space that has no memory addresses to use as pointers?
- How do we guarantee that the unique identifier we generate cannot be, either accidentally or intentionally, duplicated in TS?
- What assumptions must be met in order for the solution at this level to be successful?
- How should the unique identifier be used to identify the tuple?
- Should tuples be uniquely identifiable solely on the basis of the *uid* or is it acceptable to allow two tuples to have the same *uid* if they have different parameter signatures?
- Given that tuples comprising an LDS must have at least two *uid* fields – one to uniquely identify the tuple itself and one to refer to the tuple that represents the next node in the LDS – how do we distinguish between multiple *uid* fields within the same tuple?
- How are unique *uid* instances assigned?
- How are the semantics of the Linda *out* operation altered with respect to tuples containing *uid* parameters?
- How are the semantics of the Linda input operations – *in*, *inp*, *rd*, and *rdp* – modified with respect to tuples containing *uid* parameters?

3.1 – Overview

Given that the goal is to provide pointer semantics, and ultimately support for dynamic data structures, within the Generative Communication (GC) domain, one must first determine the necessary conditions for the existence of pointer semantics. Conventional pointers are location-addressable – a one-to-one mapping between address and the location it identifies. In GC, however, there is no one-to-one mapping between an address and the location it identifies. Data in GC are content-addressable. There is nothing, in GC, to prevent two or more tuples from matching a specific request. In such a case, one tuple is selected non-deterministically and returned to the requesting process. This fact constructs certain fundamental roadblocks for a programmer attempting to take advantage of pointer semantics to construct dynamic data structures such as linked lists. This chapter examines, in detail, the nature of these roadblocks and the solutions thereto.

3.2 – Level 1 Assumptions

For reasons that will become apparent later, in the discussion of adding support for dynamic data structures, a single assumption about the use of the enhanced TS semantics must be met in order to ensure the data integrity and structural integrity of any dynamic data structures constructed with Level 1 functionality. The critical assumption for Level 1 is that we are operating in an environment in which a single process will have exclusive access to the entire data structure in TS. Concurrent access to the data structure cannot be permitted at Level 1. There is nothing to prevent a programmer from attempting to access a Level 1 data structure with multiple processes. However, data integrity and structural integrity will likely be compromised if he or she attempts to do so. Level 1 exists to provide the foundational building blocks of support for cooperative concurrent access (Level 2) and competitive concurrency (Level 3).

3.3 – How do we achieve pointer semantics in a shared data space that has no memory addresses to use as pointers?

On the surface, the solution to this issue seems obvious. The obvious response is to construct a unique identifier that will serve in place of a memory address to uniquely identify the tuple. However, as we will demonstrate, this is easier said than done. In fact, the remaining issues at this level derive from this "obvious" solution. The scenario detailed below illustrates the nature of this problem.

A programmer constructs a list, *L* in tuple space by depositing tuples with a tag (an integer, for example) that uniquely identifies each tuple that is a member of his or her list, within the list itself. The unique tags are also used, within each tuple comprising the list, to reference the next tuple in the list. The list *L* is formally described as follows.

$L = \{t_1, t_2, \ldots, t_{(n-1)}, t_n\}$ where t_i for $i \leq n$ is a tuple of the form:

(*identifier*, [*data element*,]+ , *next node id*).

The following table describes the components of the preceding tuple.

Table 3.1 Components of the tuple

identifier	An tag that is generated and managed by the programmer such that no two identical *identifiers* exist within the context of the programmer's list.
[*data element*,]+	One or more comma-separated data elements of any of the primitive or aggregate types permitted as tuple parameters.
next node id	The *identifier* of the tuple representing the next node in the programmer's list. A '*next node id*' of -1 indicates that there are no more nodes in the list.

In addition to the tuples that comprise the list, two additional tuples are required. One tuple of the form:

LH = (*s*, *head*), where *s* is a string identifying the list and *head* is the *identifier* of the first tuple in *L*. The purpose of *LH* is to maintain a reference to the head of the linked list.

The second tuple of the form:

LIndex = (*s*, "*index*", *next available id*), where *s* is the same string used to identify the list in the tuple *LH*, "*index*" is a string literal that allows the programmer to differentiate between *LIndex* and *LH*, and *next available id* is the next available instance of the type being used as *node identifiers* (in this example it is an integer). This tuple, as the description implies, is used to keep track of the next available *identifier*. When the programmer wishes to add a new tuple to the list, he or she must *in* this tuple, use it to identify the new tuple, *out* the new tuple, then *out* the tuple *LIndex* with the *next available id* field incremented to the next available *identifier*.

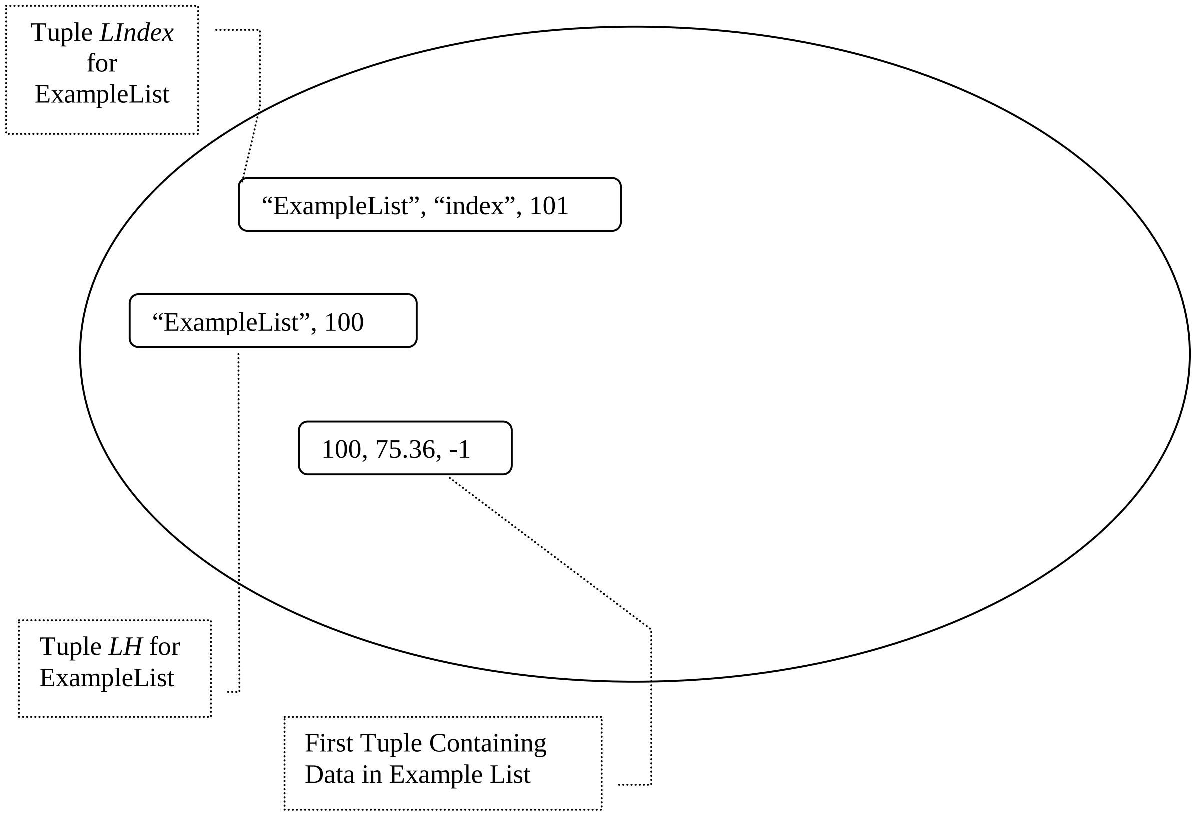

Tuple *LIndex* for ExampleList
"ExampleList", "index", 101
"ExampleList", 100
100, 75.36, -1
Tuple *LH* for ExampleList
First Tuple Containing Data in Example List

Figure 3.1 – State of tuple space following initialization for *ExampleList* and addition of 1 list node.

Figure 3.1 illustrates the state of tuple space after the example list, named "ExampleList", has been created and one node stored in tuple space. The data in the nodes of the example list will be real numbers so as to make them stand out distinctly from the metadata of the list. Continuing with the scenario, the programmer subsequently begins to traverse the list *L*. However, in the interim, another process has deposited a tuple, *FT*, that coincidently has the same parameter signature as tuples in the linked list *L*. Furthermore, the parameter of *FT* that corresponds to the *identifier* field of $t_i \in L$ is associated with a representation of the same actual value as is associated with $t_i \in L$. A problem arises because, when faced with two tuples that match the request exactly, one of the tuples is non-deterministically selected for return. Therefore, when the programmer requests the return of tuple $t_i \in L$ it is entirely possible that the tuple returned from TS will not be that tuple but rather *FT*. Figure 3.2 illustrates this situation more clearly.

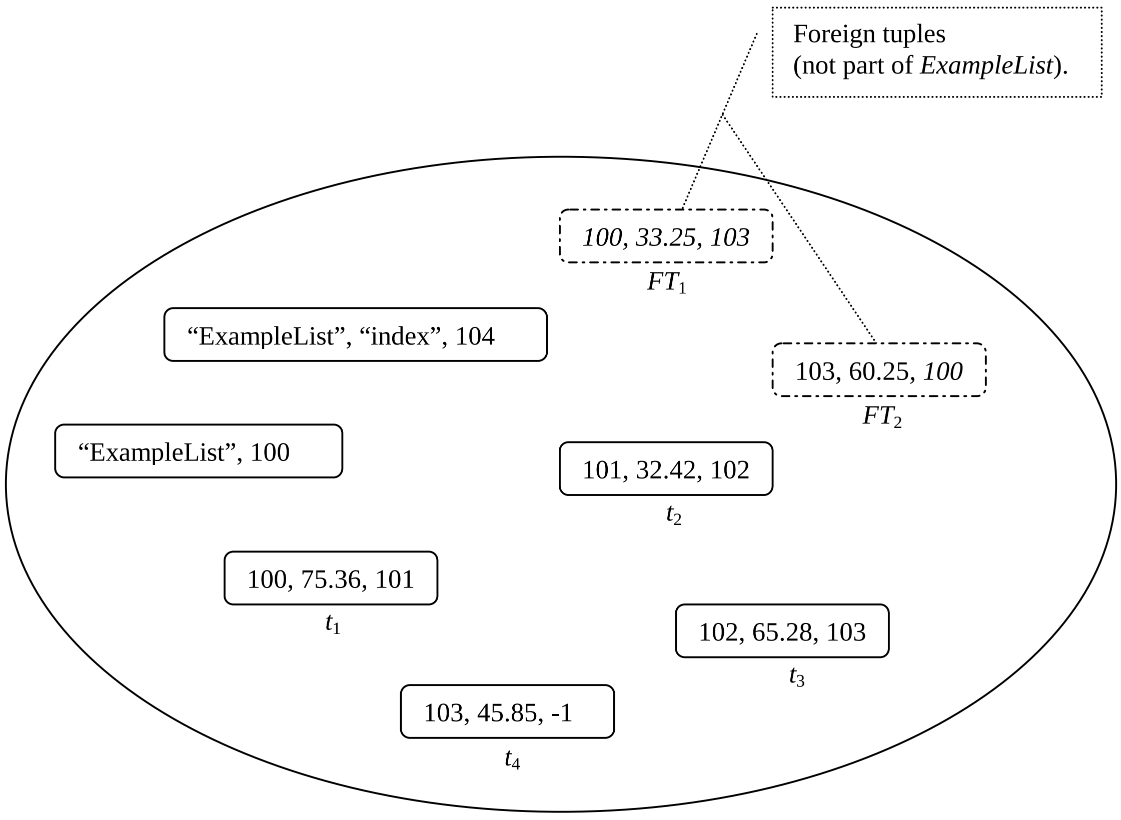


Figure 3.2 – Tuple space after multiple nodes have been added to *ExampleList*

We see, in Figure 3.2, the addition of several more nodes to *ExampleList*. Additionally, another programmer (presumably) has added the tuples FT_1 and FT_2, which, coincidentally, correspond to the format of the tuples in *ExampleList*. To fully appreciate the essence of the problem, imagine for a moment that the programmer working with *ExampleList* executes the following Linda command.

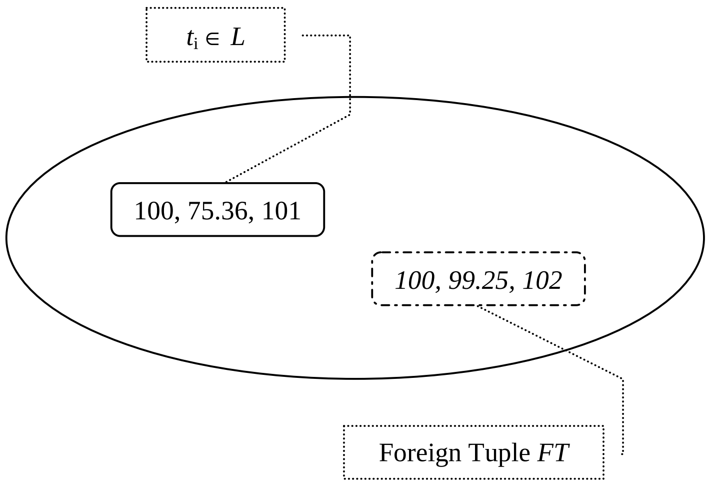
```
int    nextID;    /*integer to hold the id of next node*/
double doubleVar; /*double to hold data retrieved*/

in(100, ?doubleVar, ?nextID);
```

The programmer would, no doubt, be expecting that tuple t_1 would be returned from TS. However, because of the nondeterministic semantics of GC, it is entirely possible that tuple FT_1 could be retrieved instead. The minimum impact that would occur if tuple FT_1 were returned rather than t_1 is that the programmer receives the incorrect data for that node in the list. More dramatic deleterious effects occur if the programmer is relying on the *next node id* field of the imposter tuple in order to be able to traverse to the next node in the list. If that were to prove to be the case, then the program would end up skipping over nodes t_2 and t_3 of the *ExampleList*. The worst-case scenario would occur if TS happened to return tuple FT_2 instead of tuple t_4. If that were to happen, instead of terminating the list traversal at the last node, traversal would continue back to the beginning of the list.

3.4 – How do we guarantee that the unique identifier we generate cannot be, either accidentally or intentionally, duplicated in TS?

Virtually any identifier we generate could be mimicked by another process. Traditional GC semantics specify that any number of identical tuples can exist in TS. Furthermore, it is possible that even tuples that are substantially different, in terms of actual the value representations associated therewith, might still match a template that contains formal parameters.

The solution to the problem described in Section 3.3 is to extend GC to provide a mechanism by which a programmer can retrieve a unique tag from TS that can be used to uniquely identify his or her tuple. The central question, however, is: How can we provide a tag for which there is an explicit guarantee that no other process or programmer could coincidently produce an identical tag?



```
int    nextID;    /*integer to hold the id of next node*/
double doubleVar; /*double to hold data retrieved*/

read(100, ?doubleVar, ?nextID);
```

Figure 3.3 – Tuples that have different parameter signatures yet still cause problems.

A naïve, first pass solution is to alter GC semantics in such a way that every tuple stored in TS must have a unique signature – that is, no two tuples stored in TS can have an identical set of parameters. However, this still does not address the problem because the situation described in Section 3.3 meets this requirement, yet still leads to nondeterministic results. As illustrated by Figure 3.3, the problem in Section 3.3 occurs when the types and positions of parameters in the 'foreign' tuple, *FT*, match the types and positions of the parameters in $t_i \in L$, and only one of the associated values is identical. In that case, the parameter signature is different because n-1 of the n parameters are associated with representations of different actual values. Therefore, tuple *FT* passes muster under the 'unique signature' test.

The problem arises because the programmer building *L* is retrieving tuples using only one actual parameter and formal parameters in the rest of the template. It just so happens that the one actual parameter being used matches the value associated with the corresponding parameter in tuple *FT*, despite the fact that the parameter signatures of the two tuples are, otherwise, different. Again, the *in* statement in Figure 3.3 demonstrates this very clearly. The programmer cannot know if he or she will receive $t_i \in L$ or *FT*.

An even more naïve solution, in light of the failure of the first attempt, would be to further restrict the GC semantics so that every tuple stored in TS must have a unique signature based solely on formal parameters regardless of actual representations of values associated therewith. The folly in this idea is that, if we were to adopt such draconian measures, we create a situation in which TS is now useless for all practical purposes. We cannot even store the elements of our own list because the tuples that comprise our list are not unique on the basis of formal parameters.

The problem is addressed by adding a new data type, *uid* (for *unique identifier*), to the language. Because *uid* is a new data type we can alter GC semantics, with respect to it, without affecting the traditional GC semantics with respect to existing data types. The addition of the *uid* data type allows GC semantics to be altered so that every tuple in TS that contains a *uid* parameter must be uniquely identifiable based upon a *uid* parameter. In the abstract, one can think of TS as being initialized with the set of all possible tuples with a single *uid* parameter. When a programmer wishes to use a *uid* he or she removes one of the tuples, containing a single *uid* parameter, from TS. The programmer may then use the *uid* to uniquely brand an element of his or her data structure. The existing set of GC primitives, extended to encompass the solution proposed in this chapter is, collectively, referred to as Level 1.

3.5 – How should the unique identifier be used to identify the tuple?

From a pointer semantics perspective, the *uid* should refer to a location wherein the data are stored (Figure 3.4). To access the data, the programmer would then dereference the pointer to access the data. This option clearly raises some issues with respect to how the dereferencing operation would be accomplished. Also at issue is what effect this mechanism would have on the semantics of the GC operations such as *in*, *rd*, etc.

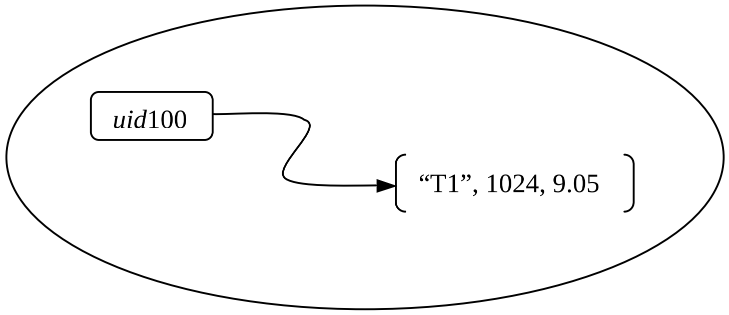


Figure 3.4 – Unique identifier usage consistent with pointer semantics

From the perspective of GC semantics, the *uid* should be a field in the tuple. To access the data, the programmer must execute either a *rd*, *rdp*, *in*, or an *inp* operation specifying the *uid* as an actual parameter in the template. The remainder of the parameters would be specified as formal parameters.

Given that we are incorporating pointer semantics into GC rather than incorporating GC semantics into pointers, we opt for the solution that more closely matches GC semantics. The benefits that might be realized by adopting the solution favoring pointer semantics do not justify the additional compromise to GC semantics that would be incurred were we to adopt that solution. Colloquially speaking, it would be like taking two steps away from GC semantics to get but one step closer to pointer semantics.

3.6 – Should tuples be uniquely identifiable solely on the basis of the *uid* or is it acceptable to allow two tuples to have the same *uid* if they have different parameter signatures?

Figure 3.5 illustrates the two alternatives. Tuples t_1 and t_2 can be distinguished from one another based upon *uid* alone. On the other hand, tuples t_1 and t_3 have the same *uid* but they can still be distinguished from one another based upon their parameter signatures. Traditional GC semantics hold that two tuples with different parameter signatures are, in fact, distinct. Therefore, in theory, there is nothing to prevent us from adopting a convention that allows the same *uid* to be used as a partial key in two or more tuples as long as the parameter signatures, combined with the *uid*, ensure uniqueness among all other tuples in TS. With respect to pointer semantics, however, a situation in which one pointer refers to two different blocks of data is utter nonsense. For this reason alone, we have sufficient reason to mandate that no two tuples share the same *uid*. However, as we will discuss in a later section of this document, there is an even more compelling reason to opt for this solution.

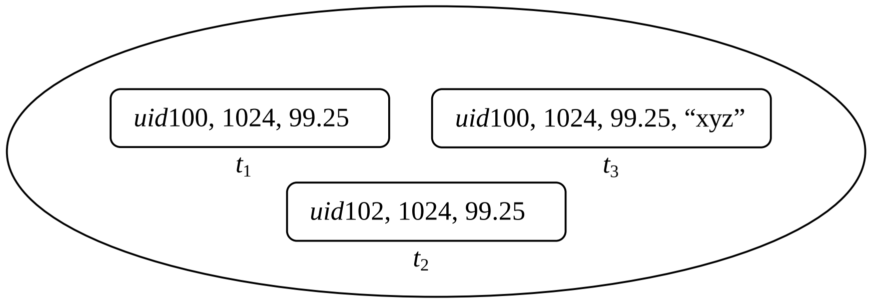


Figure 3.5 – Uniqueness based solely upon *uid* vs. *uid* and parameter signature.

We need, at this point, to call attention to the following: With previous issues we have favored the solution most congruent with traditional GC semantics choosing to yield some degree of faithfulness to pointer semantics. In the case of this issue regarding how the *uid* should be used to uniquely identify a particular tuple, we follow the opposite tack. This seeming lack of consistency is, however, completely justifiable. In previous situations, the detriment to pointer semantics benefited GC semantics. In this case, yielding to GC semantics leads to a serious inconsistency with pointer semantics. However, given that we are outside the realm of normal GC semantics anyway, insofar as we are mandating tuples that can be uniquely identified, we lose nothing with respect to GC semantics by opting for the solution more closely in line with pointer semantics.

3.7 – Given that tuples comprising an LDS must have at least two *uid* fields – one to uniquely identify the tuple itself and one to refer to the tuple that represents the next node in the LDS – how do we distinguish between multiple *uid* fields within the same tuple?

This is a critical issue because the tuplespace manager (TSM) must be able to detect, and prevent, attempts to use the same *uid* to identify more than one tuple. Clearly, however, the same *uid* must appear in more than one tuple. In one tuple (Tuple A), it is used to uniquely identify that tuple. The identifier must appear in another tuple (Tuple B) to serve as a reference from Tuple B to the tuple it uniquely identifies (Tuple A).

One solution would be to arbitrarily fix the location, within an LDS tuple, of the *uid* that is to serve as the identifying feature of that tuple. For example, we could mandate that the first parameter of the *uid* containing tuple must uniquely identify that tuple. All other *uid* fields within that tuple are taken to refer to other tuples. Another variation on this theme would be to place the *uid* that uniquely identifies a particular tuple in the first parameter position. The data are encased in a structure and the references to other tuples are either in the structure or are located in the tuple in parameter positions subsequent to the data containing structure.

A completely different option is to use a semantic tag to specify which of the *uid* fields, contained within a particular tuple, uniquely identify that tuple. This option makes the position of the *uid* fields within the tuple, relative to other parameters contained therein, irrelevant.

The “fixed location” option might appeal to some because it conjures images of the “pointer to a structure” semantics normally associated with traditional pointers. However, we select the “semantic tag” option because it parallels, more closely, the semantics of GC. Traditional GC semantics do not assign meaning to tuple parameters based upon their location within the tuple. By the same token, however, the inclusion of a semantic tag, to distinguish the unique identifier from other *uid* fields, also represents a departure from traditional GC semantics. In the case of the “fixed location” solution, though, the deviation from traditional GC semantics yields less flexibility for the programmer and significantly alters GC semantics. The “semantic tag” option, however, preserves maximum flexibility for the programmer while only slightly modifying GC semantics. We opt, therefore, for the “semantic tag” solution. The *caret* (‘^’) symbol is added to the *uid* parameter that uniquely identifies the tuple, designating it as the *primary uid* of the tuple. No two tuples can exist in TS with the same *primary uid*. The TSM enforces this restriction on the use of the *uid* data type. The code segment below shows the use of the semantic tag to designate the *primary uid* of the tuple being stored in TS. We assume the correct a priori declaration and initialization of the variables `uniqueID` and `nextUniqueID`.

```
out(^uniqueID, 14, nextUniqueID);
```

3.8 – Requesting *uid* Value

Traditional pointers are instantiated by requesting, from the runtime environment of a process, a block of memory. The request is answered with the address of the location of that block of storage. In the case of GC, we do not have storage to be reserved. Nonetheless, the *uid* must come from somewhere. The TSM is the entity responsible for assigning the *uid*. The question is, though, by what mechanism are they requested?

Ideally, the *uid* assignment/request process should fit into the GC framework as naturally as possible. To accomplish this goal, we begin by conceptually viewing TS as having been initialized with one-tuples containing all possible *uid* instances. In order to retrieve a new *uid*, the programmer must declare a variable of type *uid* and initialize it to *NULL*. The programmer then executes the *inp* or the *in* operation with the previously declared *uid* variable as its only formal parameter. If the operation is successful, the *uid* variable will become associated with a representation of a non-null value interpreted to be an instance of type *uid*. The programmer can then use the *uid* to store a uniquely identifiable tuple.

The *inp* operation was chosen as the preferred mechanism for obtaining a *uid* from the because its normal semantics most closely match the semantics of its application in this situation. The normal semantics of the *inp* operation are that a tuple matching the set of specified parameters is non-deterministically chosen from the set of all tuples matching the specified template. The tuple is then removed from TS and, as such, is no longer available to other processes. When a programmer requests a new *uid* from the TSM, a one-tuple containing a *uid* is non-deterministically selected, from all matching tuples in TS, and returned. It is not available to other processes until the process that requested the tuple returns it to TS using an *out* operation. Given that the *uid* tuple is selected non-deterministically, programmers must not make any assumptions regarding the sequence in which the singleton tuples containing *uid* instances will be retrieved from TS.

The *in* operation can also be used to request a *uid* from TS. The semantics of the *in* operation closely match the semantics of the *inp* operation. The difference is that, if no matching tuple exists in TS, the process executing the *in* operation blocks until a matching tuple arrives in TS. The *inp* operation, on the other hand, does not block but instead returns with a failure code. Given the parity of semantics between these two operations, there is virtually no justification for disallowing the *in* operation from being used to retrieve a *uid*.

In order to ensure the uniqueness of the *uid*, the one-tuple containing that *uid* must be removed from TS. Therefore, it is necessary to disallow the use of the *rd* or *rdp* operations with a single *uid* formal parameter. To allow the use of these operations would compromise the integrity of the *uid* contract.

3.9 – The *out* Operation

The *out* operation behaves, as traditional GC semantics would dictate, with some notable exceptions. First, all *uid* containing tuples must have a *primary uid*. The *primary uid* is specified through the use of a semantic tag in order to avoid the necessity to restrict the position of the *primary uid* in the tuple. The code segment below illustrates the process of retrieving a new *uid* from TS and then storing a tuple, in TS, that is uniquely identified by that *uid* value.

```
uid uniqueID = NULL;
uid nextUniqueID = NULL;

inp(?uniqueID);
out(^uniqueID, 14, nextUniqueID);
```

Notice that the *caret* symbol preceding the *uid* variable *uniqueID* indicates that the *uid* value is, in fact, the *primary uid* of the tuple whereas *nextUniqueID* is just a reference to another tuple. No two tuples in TS may be identified by the same *primary uid*. The *primary uid* of one tuple may be stored in another tuple as long as that tuple has its own *primary uid*.

Second, from the programmer's perspective, TS is initialized with a set of one-tuples, each tuple of which contains one member of the set of all possible *uid* values. This means that a *uid* cannot be specified as a *primary uid* until the one-tuple containing that *uid* has first been removed from TS. This restriction follows as a natural consequence of the rule that states that no two tuples may be identified by the same *primary uid*.

The final difference between Level 1 and traditional GC occurs when a programmer executes the *out* operation with a single *uid* actual parameter. When this happens, the *uid* has been returned to TS just as any other tuple would be. Also, like any other tuple, it is now available for another programmer to retrieve. The difference is that there is a contract, in the case of the *uid*, which guarantees the uniqueness of the *uid*. Because of this, no *uid* value can be returned to TS if that value is the *primary uid* of another tuple already in TS. This is, again, a logical consequence of the restriction that no two tuples with the same *primary uid* can be stored in TS. The reason for this is that, when a programmer executes an *out* operation on a tuple with a single *uid* actual parameter he or she is, in essence, storing that tuple in TS with its own value as the *primary uid* of the tuple.

3.10 – The Input Operations (*rdp*, *inp*, *rd*, and *in*)

The standard TS operations *rdp*, *inp*, *rd*, and *in* are used to access data stored in TS. By using a known *uid* value as an actual parameter to any of these operations, the programmer can be assured that no more than one tuple will match. Consistent with traditional GC semantics, the Level 1 versions of the *rd* and *rdp* operations return the tuple data to the calling process while leaving the tuple in TS. The Level 1 versions of the *in* and *inp* operations return the tuple data to the calling process and remove the tuple from TS. The difference between the *rdp* and *rd* is that *rdp* is a non-blocking operation.

In other words, if the requested tuple is not in TS, the *rd* operation will block indefinitely waiting for a matching tuple. The *rdp* operation will not block but will rather fail and return. The same is true of the *inp* and *in* operations.

Retrieving a tuple via its *uid* value is roughly analogous to dereferencing a normal pointer. However, there is a divergence from traditional pointer semantics that cannot be avoided. This divergence is illustrated by the following example.

```
1) typedef struct exampleStruct {
2)    int intData1;
3)    int intData2;
4) } exampleType;

5) exampleType exampleInstance;
6) exampleType *examplePtr = &exampleInstance;

7) examplePtr ->intData1 = 10;
8) (*examplePtr).intData2 = 15;
```

In this example, lines 7 and 8 demonstrate two different mechanisms for accessing individual elements of the structure. There is no equivalent mechanism within the GC context. Tuples containing aggregate data types must be retrieved in their entirety into the local address space of the process. Once the tuple has been retrieved, the elements of the aggregate type can be accessed individually. This divergence with traditional pointer semantics is necessary to maintain consistency with traditional GC semantics, which do not provide for retrieval of partial tuples or part of an aggregate that is an element in a tuple.

Chapter 4 – Pointer Support for GC: Level 2

This chapter examines the construction and manipulation of LDS in TS within an environment in which processes concurrently accessing nodes of the LDS cooperate with one another.

This chapter addresses the following issues:

- We showed in Chapter 3, that the *uid* data type is necessary to support LDS in TS. The central question, at this point, is whether or not the *uid* data type is sufficient to support concurrent manipulation of LDS in TS?
- What additional support must be added to GC to facilitate concurrent manipulation of LDS constructed using the *uid* data type?
- What assumptions must be met in order for the solution at this level to be successful?
- Are concurrent operations on LDS in TS prone to deadlock? If so, how can this be prevented?
- How is mutual exclusion achieved and what are the implications for processes traversing the LDS?
- How is the "Reader's Problem" resolved?
- What changes to the cooperative protocol are necessary to accommodate the *lock* and *unlock* operations?
- The contract of the lock operation guarantees that a locked tuple will be returned to TS. Given that fact, can a reading process, upon encountering a locked tuple, issue a non-predicate rd operation on that tuple without fear of indefinite blocking?
- Why can we not ensure the success of a non-predicate rd operation by allowing it to execute as soon as a locked tuple is unlocked, thereby eliminating the need for a reading process to execute a "hard wait?"

4.1– Overview

The *uid* data type introduced at Level 1 is necessary and sufficient to support sequential access to an LDS in TS. However, imposing the draconian restriction against shared access to the LDS begs the question: “Why bother with tuplespace, and the overhead associated with calls thereto, if we cannot share the data structure in question with other processes?” In fact, it would be terribly inefficient to use TS as a storage medium for an LDS that is only going to be accessed by a single process. The local address space provides a much more suitable medium for that purpose. In this chapter, we open Pandora’s box by allowing concurrent access to LDS constructed using Level 1 primitives. We examine the resultant problems and develop the solutions to those problems. We narrow the scope of the problem at this level by assuming that processes operating concurrently on an LDS do not compete for access to the LDS, but instead cooperate.

4.2 – Level 2 Assumptions

The issues and solutions outlined in this chapter guarantee the structural integrity and data integrity of the LDS so long as one fundamental assumption is met. The contract at Level 2 assumes that the environment in which the LDS is constructed, accessed, and updated, is a cooperative concurrent environment in which all processes adhere to the protocol developed in the sections that follow.

4.3 – Is the *uid* data type sufficient to support concurrent manipulation of LDS in TS?

Having naively removed the prohibition against concurrent access to the LDS in TS, we examine a pair of scenarios that will help us determine if we have chosen wisely or not.

The first scenario consists of two processes simultaneously attempting to increment the value field of a linked list tuple. We begin with the linked list *ExampleList*, the initial state of which is shown in Figure 4.1.

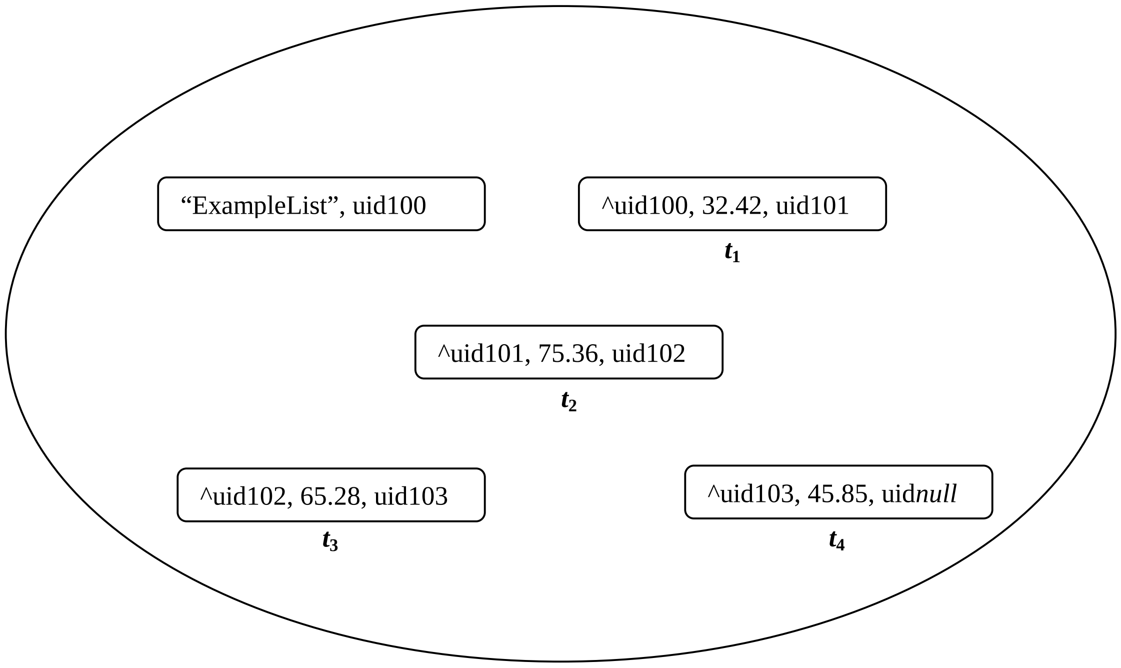


Figure 4.1 – Initial state of linked list *ExampleList*.

The notation used in Figure 4.1 is as follows. The reference to the first node in *ExampleList* is maintained in the tuple containing a string identifying the name of the list and the *primary uid* of the first node in the list. In each tuple $t_n \in$ *ExampleList* the notation `^uid`*x*, where *x* is a unique number, represents the *primary uid* of that tuple. A reference to another tuple is denoted by `uid`*y*, where *y* is a unique number and `uid`*y* is the *primary uid* of the next tuple in the list. The data element in each tuple consists of a real number with two digits shown following the decimal place. This is an arbitrary decision. In reality, of course, the data in each tuple of the linked list can be any data type, either primitive or aggregate, allowed as an element in a tuple within the traditional GC framework. Additionally, tuples can contain more than one data field at the sole discretion of the programmer. However, the nature of linked lists dictates that all of the tuples comprising a given list must be consistent with respect to position and type of each element in the tuple.

The following table (Table 4.1) illustrates the relative timeline of the events associated with the updates by processes P_1 and P_2. The description of the scenario begins at time

Θ_3, at which both processes have read tuple t_2 and decided upon it as their common target for the update.

Table 4.1 – Timeline of events associated with simultaneous update of an LDS tuple.

Time Θ_n	Process P_1	Process P_2
Θ_1	rdp(t_2), value ← 75.36	–
Θ_2	Decide to update t_2	rdp(t_2), value ← 75.36
Θ_3	Increment t_2.value, new value ← 76.36	Decide to update t_2
Θ_4	–	Increment $t_{2.}$.value, new value ← 76.36
Θ_5	–	inp(t_2)
Θ_6	–	out(t_2), t_2.value ← 76.36
Θ_7	inp(t_2)	–
Θ_8	out(t_2), t_2.value ← 76.36	–

In Table 4.1, process P_1 and process P_2 perform two separate updates on tuple t_2, each one adding 1 to the initial value. The initial value of the data element in t_2 is 75.36. After two processes increment the value by one, the expected value is 77.36, however, because of the concurrent update, a classic race condition ensues and one of the updates is lost. The result of this sequence of events yields a value of 76.36. In other words, concurrent access, in this case, leads to a violation of data integrity.

In the second scenario, process P_1 adds a node to the LDS, and process P_2 deletes a node from the list. Figure 4.2 represents the initial state of the LDS. In the scenario, process P_1 inserts a node between t_2 and t_3, while simultaneously, process P_2 deletes t_3. Table 4.2 details the timeline of events associated with these operations beginning at time Θ_{01}, the point at which process P_1 reads t_2 and decides to insert a new node immediately following it.

Table 4.2 – Timeline of events associated with simultaneous update of an LDS tuple.

Time Θ_n	Process P_1	Process P_2
Θ_{01}	rdp(t_2)	rdp(t_2)
Θ_{02}	Decide to insert after t_2	rdp(t_3)
Θ_{03}	Construct t_{new}	Decide to delete t_3
Θ_{04}	t_{new}.next $\leftarrow$ t_2.next	inp(t_2)
Θ_{05}	t_2.next $\leftarrow$ t_{new}	inp(t_3)
Θ_{06}	–	t_2.next $\leftarrow$ t_3.next
Θ_{07}	–	out(t_2)
Θ_{08}	inp(t_2)	–
Θ_{09}	out(t_{new})	–
Θ_{10}	out(t_2)	–

It might not be apparent from the timeline alone, but process P_2 updates t_2.next to refer to tuple t_4 in preparation to delete tuple t_3. This update is subsequently lost when process P_1 stores its copy of tuple t_2 using the *out* operation. This is clearly illustrated in the following sequence of figures.

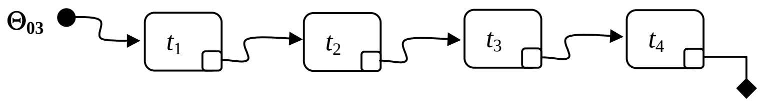


Figure 4.2 – Actual state of LDS in TS at time Θ_{03}.

Figure 4.2 shows the actual state of the LDS in TS at time Θ_{03}. Process P_1 has already started to construct the new tuple assuming the state of the LDS is as it is represented in Figure 4.2. Figure 4.3, below, illustrates the picture that process P1 has of the state of the list at time Θ_{07}.

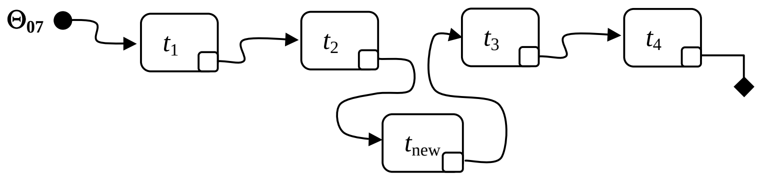


Figure 4.3 – State of LDS as perceived by process P_1 at time Θ_{07}

In its local address space, process P_{01} has constructed the new node t_{new} and updated links as shown in Figure 4.3. However, process P_2 completes its deletion of tuple t_3 at time Θ_{07}. The deletion of tuple t_3 leads to the actual state of the LDS in TS illustrated by Figure 4.4.

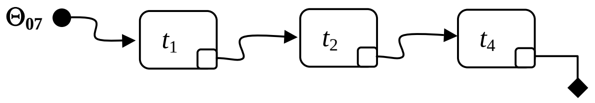


Figure 4.4 – Actual state of LDS following modification by process P_2 at time Θ_{07}.

Process P_1 constructs a link from the new node it is creating, t_{new}, to tuple t_3. However, tuple t_3 is deleted by process P_2. Therefore, when P_1 stores the tuple t_{new} and the updated tuple t_2 in TS, a lost update occurs. In this case, as we see in Figure 4.5, the lost update of the change process P_2 made to tuple $t_{2,}$ to bypass the deleted tuple, leads to a violation of structural integrity.

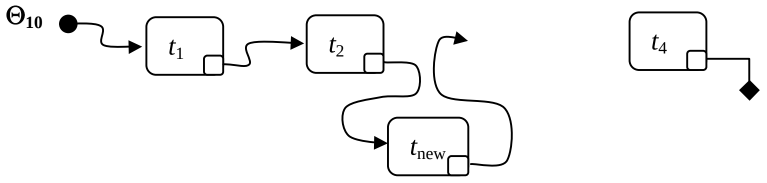


Figure 4.5 – Inconsistent actual state of LDS following modification by process P_1 at time Θ_{10}.

Two scenarios are presented in this section. The result, in both cases, is a lost update. In one case, the lost update leads to a violation of data integrity. In the second scenario, the lost update leads to a violation of structural integrity – yielding both a dangling pointer and a lost reference to the last node in the list.

4.4 – What additional support must be added to GC to facilitate concurrent manipulation of LDS constructed using the *uid* data type?

It is clear, from the examples above (Section 4.3), that the *uid* type is not sufficient by itself to support concurrent manipulation of LDS in TS. However, easily contrived examples demonstrate that concurrency can be achieved safely using the *uid* data type in conjunction with the proper sequence of operations. The critical issue here, with respect to threats to structural integrity and data integrity, is that processes operating concurrently upon an LDS in TS must have a consistent view of the state of the LDS in TS. Without an accurate snapshot of the current state of the LDS in TS, processes cannot ensure that the actions they take will not lead to deleterious results. To address this problem we introduce a cooperative protocol that ensures each process is aware of inconsistencies between its own view of the LDS and the state of the LDS, as it exists in TS.

4.4.1 – Inserting a node

To insert a node, a process traverses the list, using the rdp operation, until the insertion point is located. The insertion point is the node after which the new node is to be inserted. The process then removes the insertion point tuple from TS using the *inp* operation. The results of the *inp* operation are compared to the results of the previous *rdp* operation executed during the traversal. If the tuple remains a suitable insertion point according to the criteria being used by the programmer, the tuple representing the new node is constructed. The next pointer of the new tuple is set to refer to the tuple referred to by the insertion point tuple. The next pointer of the insertion point tuple is then set to refer to the new tuple. The new tuple is then stored in TS using the *out* operation. Finally, the insertion point tuple is returned to TS using the *out* operation and the insertion is complete.

In order to show the insertion process more clearly we begin with the state of the LDS in TS as shown in Figure 4.2. The goal is to insert a new node into the LDS. We follow the insertion process through from start to finish.

Step 1 – Identify the insertion point tuple. For our purposes, the insertion point tuple is the tuple representing the node immediately after which the new node will be inserted. Figure 4.6 illustrates the example LDS in TS with the insertion point tuple identified.

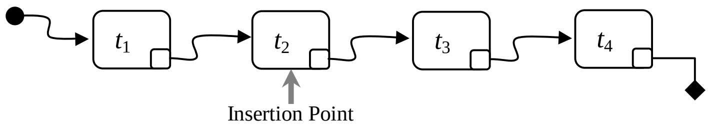


Figure 4.6 – Example LDS with insertion point tuple identified.

Step 2 – Remove the insertion point tuple from TS. As shown in Figure 4.7, the insertion point tuple is removed from TS so that the pointers can be updated to incorporate the new node. At this point, the insertion point tuple must be compared to its state at the time the decision was made to insert the new node as its immediate successor. If the data or pointers have changed, the process must return the insertion point tuple to TS and start the process again in order to ensure the data integrity and the structural integrity of the LDS.

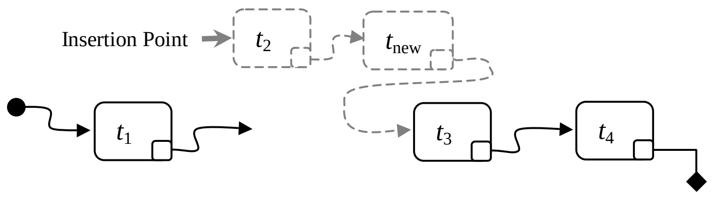


Figure 4.7 – Example LDS with insertion point tuple removed from TS.

Step 3 – Construct the new tuple, link it to the tuple referred to by the next pointer of the insertion point tuple, and make the insertion point tuple refer to the new tuple. Figure 4.8 shows the insertion point tuple and the new tuple with their pointers updated outside of TS.

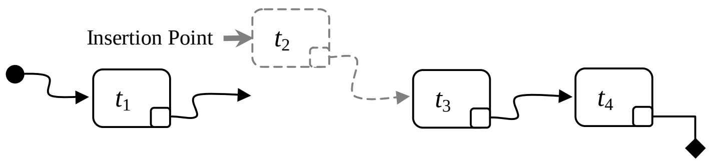


Figure 4.8 – Example LDS with insertion point tuple linked to new tuple outside TS.

Step 4 – Place the new tuple and the insertion point tuple into TS. As illustrated by Figure 4.9, the final step in the insertion process is to store the new tuple in TS, followed by the insertion point tuple. At that point, the insertion process is complete, the new node is successfully inserted, and the structural integrity of the list is preserved.

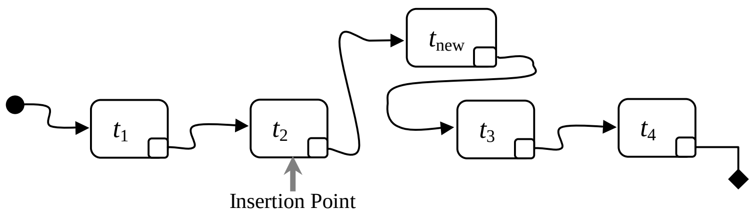


Figure 4.9 – Insertion point tuple and new tuple returned to TS completing insertion.

4.4.2 – Deleting a node

To delete a node, a process traverses the list until the deletion point and deletion target are found. The deletion target is the tuple representing the node to be deleted. The deletion point is the node immediately preceding the deletion target. The process then removes the deletion point tuple from TS using the *inp* operation. The results of the successful *inp* operation must be compared to the results of the previous *rdp* operation, executed during the traversal, to ensure that the deletion point remains a viable deletion point. The next step is to remove the deletion target tuple from TS using the *inp* operation. As with the deletion point, the contents of the deletion target tuple, returned by the *inp* operation, must be examined to ensure that they still meet the programmer's criteria for the deletion target. If both tuples meet the programmer's criteria, then the next pointer of the deletion point tuple is set to refer to the tuple to which the next pointer of the deletion target tuple refers. The deletion point tuple is then returned to TS using the *out* operation, completing the deletion process.

Beginning with an example LDS in TS, as shown in Figure 4.2, we present the following example of deletion under the cooperative protocol to clarify the process.

Step 1 – Identify the deletion target tuple and the deletion point tuple. In this context, the deletion target tuple refers to the actual tuple to be permanently deleted from TS. The deletion point tuple is the immediate predecessor of the deletion target tuple. Figure 4.10 shows the example LDS in TS with the deletion point tuple and deletion target tuple identified.

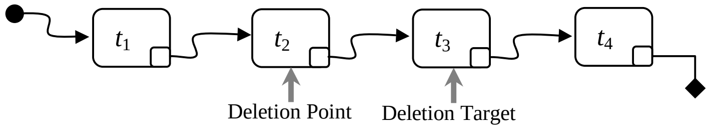


Figure 4.10 – LDS with deletion point tuple and deletion target tuple identified.

Step 2 – Remove the deletion point from TS and verify that it remains unchanged since the decision was made to delete its successor. If it remains acceptable, remove the deletion target tuple from TS and verify that it is still acceptable as the deletion target. If either node has been changed, then the nodes must be returned to TS and the deletion process must be restarted from the beginning. Figure 4.11 illustrates the state of the LDS after this step is performed.

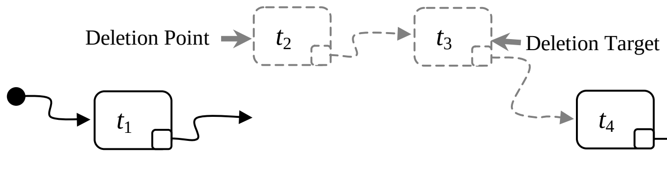


Figure 4.11 – LDS with deletion point tuple and deletion target tuple removed from TS

Step 3 – Update the next pointer of the deletion point tuple to bypass the deletion target. At this point in the process, both the deletion point tuple and the deletion target tuple are out of TS. Additionally, both tuples are verified as unchanged since the decision was made to delete the node in question. As Figure 4.12 illustrates, the next pointer of the deletion point tuple is set to refer to the successor of the deletion target, effectively removing the deletion target tuple from the chain.

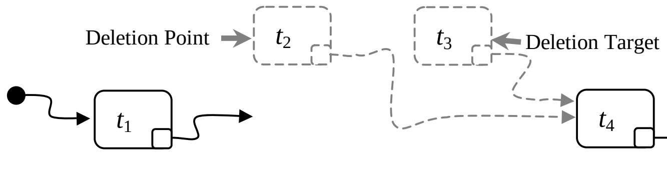


Figure 4.12 – Deletion point tuple next pointer updated to bypass deletion target.

Step 4 – Return the deletion point tuple to TS. Figure 4.13 shows the state of the LDS in TS after this step is complete. The deletion target is no longer in the LDS and the structural integrity of the list remains intact.

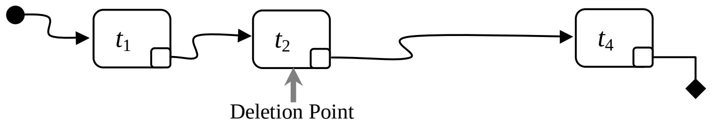


Figure 4.13 – Deletion point tuple returned to TS to complete deletion.

4.4.3 – Updating a node

To update a tuple the process first traverses the list, using the *rdp* instruction, until it reads the tuple matching the programmer's criteria. This tuple is the update target tuple. The update target tuple is removed from TS using the *inp* operation. The results of the successful *inp* operation are then compared to the results of the previous *rdp* operation to ensure that the tuple contents still meet the programmers criteria for the update target. The process then changes the data in the tuple and returns it to TS using the *out* operation, thereby completing the update procedure.

The following example demonstrates the cooperative update protocol. The goal of the update, in this case, is to increment, by one, the data value of the tuple containing 75.36.

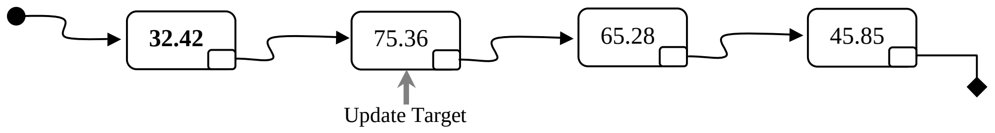


Figure 4.14 – Example list with the update target identified.

Step 1 – Identify the update target. The update target is the tuple that is to be altered. Figure 4.14 illustrates the state of the example LDS, in TS, with the update target tuple identified.

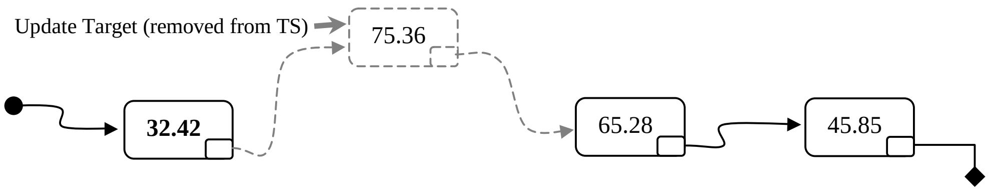


Figure 4.15 – Example list with the update target removed from TS.

Step 2 – Remove the update target from TS. With the update target identified, the process removes the update target from TS and verifies that it has not been altered since the decision was made to update it. If it remains a viable update target, according to the programmer's criteria, then the update can proceed. Otherwise, the update target tuple is returned to TS and a new update target must be found. Figure 4.15 shows the update target tuple removed from TS and prepared for the update.

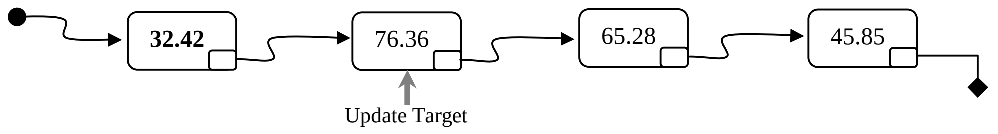


Figure 4.16 – Example list with the update target value incremented by one.

Step 3 – Alter the data. After the update target tuple is removed from TS and its viability is verified, the data can be altered as per the intent of the programmer. Figure 4.16 shows the update target tuple with the data incremented from 75.36 to 76.36.

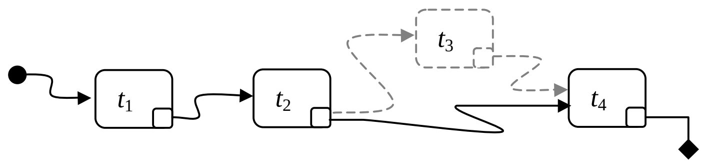


Figure 4.17 – Example list with the update target returned to TS, completing the update.

Step 4 – Return the update target tuple to TS. The final step in the update protocol is to return the updated tuple to TS so that other processes can access it. Figure 4.17 illustrates the state of the LDS, in TS, after the successfully updated tuple has been returned to TS, thereby completing the update process.

4.5 – Are concurrent operations on LDS in TS prone to deadlock? If so, how can this be prevented?

The non-predicate Linda operations – *in*, and *rd* – are extremely prone to deadlock. If no tuple in TS matches the template specified by a process executing one of these operations, the operations block until a matching tuple is placed into TS by a concurrent process. The following scenario illuminates the danger the blocking semantics pose for processes operating on LDS in TS.

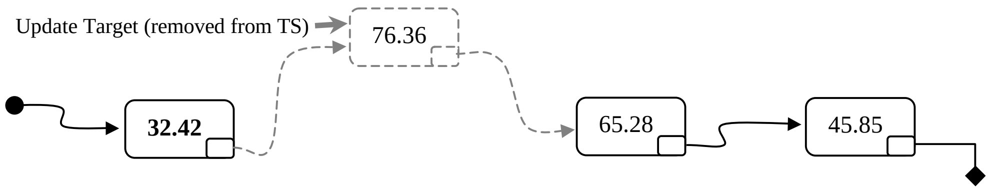


Figure 4.18 – Example list with deleted node and links shown.

Figure 4.18 shows an example LDS in TS. Assume that a process, P_1 is traversing the list. Process P_1 has just read tuple t_2, and is preparing to continue to read tuple t_3. However, before P_1 completes the move, process P_2 deletes tuple t_3. Because t_3 has been removed from TS, when P_1 attempts to read t_3 using a blocking *rd* or *in* operation, the process blocks. If a tuple identical to tuple t_3 is never returned to TS, then process P_1 is blocked indefinitely. The problem is avoided by using the non-blocking predicate operations *inp* or *rdp*.

Of course, deadlock can also arise from a chain of circular dependencies. Consider Figure 4.18, once again, as we present the following scenario. Process P_1 intends to perform an operation on tuple t_1 and t_2. For whatever reason, the process removes t_2 from TS first using an *inp* operation. In the mean time, process P_2 intends to operate on the same two tuples. It, however, removes t_1 first. Both processes now hold a resource sought after by the other process. If neither process relinquishes control of the tuple it is holding, then deadlock will result. This problem arises because the processes are allowed to "hoard" resources while waiting for more resources to become available so that they can complete their task. The solution to this problem is to carefully construct the cooperative protocol so that instances of this "hoarding" behavior do not occur.

4.6 – How is mutual exclusion achieved and what are the implications for processes traversing the LDS?

When a process requires mutually exclusive access to a node in the LDS, it removes the tuple, representing that node, from TS using the *inp* operation. The relative ease with which mutual exclusion is achieved can, however, lead to a false sense of security. An interesting issue arises from this seemingly innocuous application of normal GC semantics.

The cooperative protocol eliminates threats to LDS integrity. As the cliché states, however, every rose has its thorn. The thorn, in this case, may cause problems when a process is traversing the list using the *rdp* operation to access each tuple in the list. The reason for the traversal is irrelevant. The process could be reading nodes looking for an insertion point, looking for a node to update, or simply traversing the list to display its contents. Regardless of the reason for the traversal, a problem exists for reader processes. As a reader process attempts to follow a link from one LDS tuple to the next, it is possible that the tuple in question will not be in TS at the time. When this situation arises, two possibilities exist: either the tuple has been removed from TS for modification and will be returned to TS; or the tuple has been permanently removed from TS.

The difficulty for the reader process is that it has no way to discriminate between the two possibilities. In that former case, the reader process should retry the read until the tuple is returned to TS. In the later case, the reader process should back up to the preceding node and follow the new link. The following scenario illustrates the case in which the tuple is not returned to TS.

Figure 4.19 shows the initial state of the LDS in TS as processes P_1 and P_2 begin traversal.

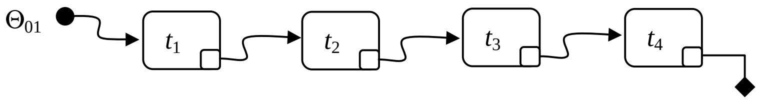


Figure 4.19 – Actual state of LDS in TS at time Θ_{01}.

Process P_1 is simply traversing the list whereas process P_2 is looking for a target to delete. Table 4.3 shows the sequence of events that lead up to the problem.

Table 4.3 – Timeline of events surrounding problematic LDS traversal.

Time Θ_n	Process P_1	Process P_2
Θ_1	rdp(t_1)	rdp(t_1)
Θ_2	–	rdp(t_2)
Θ_3	–	Decide to delete t_2
Θ_4	–	inp(t_1)
Θ_5	–	inp(t_2)

Θ_6	–	t_1.next ← t_2.next
Θ_7	–	out(t_1)
Θ_8	inp(t_2)	–

At time Θ_1, process P_1 reads tuple t_1. This means that P_1 has a view of the LDS in which t_2 is the successor to t_1. However, at time Θ_5, P_2 removes tuple t_2 from TS, effectively deleting it. At time Θ_8, P_1 attempts to read the t_2 based upon its incorrect view of the LDS. Figure 4.20 shows both the actual state of the LDS at time Θ_{08} and, in dashed lines, the incorrect picture P_1 has of the LDS.

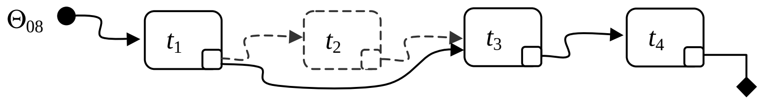


Figure 4.20 – Actual state of LDS at time Θ_{08} with P_1 perceived state shown (dashed lines).

From Figure 4.20, it is clear that P_1 has a problem. Between the time it determined that t_2 is the successor to t_1 and the time that it attempts to read t_2, process P_2 deleted t_2. When P_1 executes the *rdp* operation, the operation will fail because the requested tuple is not in TS. Therefore, the question becomes: "On what course of action should the reader process embark upon encountering this situation?"

Certainly one possible action is to loop until the tuple returns, thereby allowing the operation to succeed. The disadvantage of this option is that the tuple might not be returning to TS at all, in which case, the loop would be infinite. A second course of action is to back up to the predecessor of the target tuple and reread it. To envision the difficulty with this option, ask yourself: "What happens if the predecessor has been removed from TS when I attempt to reread it?" It does not stretch the imagination to see that, in a highly volatile list, a process could be stuck repeatedly backing up all the way to the first node in the list, ad infinitum. We dub this dilemma "The Reader's Problem."

4.7 – Resolving the Reader’s Problem

The essence of the reader’s problem, described in Section 4.6, is that the return value from the *rdp* operation does not provide sufficient information to the reader process. The result is a Boolean indicating success or failure of the predicate operation. However, three states actually need to be represented: success; failure because the tuple is gone forever; or failure because the tuple is temporarily absent from TS. The simple solution is to refine the granularity of the return value such that it accurately reflects the true status of the tuple. To that end, we modify the semantics of the predicate input operations (*rdp*, and *inp*) to return one of three status codes instead of a Boolean.

With these modifications in place, the reader process can determine, based upon the status code returned by the *rdp* and *inp* operations, which of the two alternative recovery paths, discussed in Section 4.6, it should follow to continue its list traversal. The next logical question becomes: “How does the TSM determine whether an LDS tuple is permanently removed from TS or whether it is going to be returned to TS?”

For the TSM to be able to generate the correct return code for the *rdp* and *inp* operations a mechanism must be included by which a process can indicate its intentions with respect to removing a tuple from TS. Currently no such mechanism exists. One method for providing this functionality would be to alter the *inp* and *in* operations to require the use of a flag to indicate the intentions of the programmer. We reject this technique, however, because it significantly alters the semantics of these operations with respect to non-LDS tuples. The better alternative is to include a new operation that allows the programmer to signal his or her intentions. The new operation is then included in the cooperative protocol, thereby achieving the desired results without altering the operations in question with respect to non-LDS tuples.

4.8 – The *lock* and *unlock* Operations

In Section 4.7, we motivate the need for an additional operation allowing a programmer to signal his or her intentions when removing an LDS tuple from TS. The new operation is the *lock* operation. When a process locks a tuple using the *lock* operation, it is reserving the tuple for its exclusive use. At the same time, the process is also explicitly indicating that the tuple will return to TS. When a second process attempts to access the locked tuple using the *rdp* or *inp* operation, the operation will fail with the return code LOCKED. The reader process can then wait for the tuple to return to TS. If the tuple has been removed from TS and no lock has been placed on the tuple, then the operation will fail with the return code REMOVED. The reader process can infer that the tuple has been permanently removed from TS. If the tuple exists in TS and is not locked, then the *rdp* or *inp* operation returns the data, via its formal parameters, and the return code SUCCESS. Finally, when the locking process is finished with a locked tuple and has returned the tuple to TS, it must unlock the tuple by executing the *unlock* operation to make it available to other processes.

The syntax of the *lock* operation has parity with the other GC operations. The *lock* operation requires that the programmer specify a template consisting of a combination of formal and actual parameters that match a tuple in TS. The *lock* operation is non-blocking. If the target tuple is not found, then the operation will fail with the code REMOVED. If a matching, unlocked tuple is found, then the operation will return SUCCESS. If the matching tuple is already locked, the *lock* operation will fail and return LOCKED so that the process knows that it must wait to lock the tuple. Table 4.4 summarizes the return codes of the *lock* operation and their semantics.

Table 4.4 – Return codes of the *lock* operation and their semantics.

Return Code	Semantics
SUCCESS	The process has successfully locked the tuple.
LOCKED	The tuple is already locked – it must be unlocked prior to being locked again.
REMOVED	The tuple has been removed from TS without being locked indicating that the tuple will not be returned to TS.

The syntax of the *unlock* operation requires the programmer only specify the *primary uid* of the tuple that is to be unlocked. The *unlock* operation can be called by any process, not just the process that called the corresponding *lock* operation on a particular tuple. It is assumed that, since processes are cooperating at Level 2, that a process would not maliciously unlock a tuple that was locked by another process. If the tuple is not locked and a process attempts to unlock it, the operation returns NOT_LOCKED so the process is aware of the error. If a process does not return a tuple to TS prior to attempting to *unlock* it, then the operation will return REMOVED so that the programmer is aware of the error. If the tuple is not found for any other reason (e.g., an invalid *uid* value is specified) then *unlock* operation would also return REMOVED. Barring any of the aforementioned situations, the *unlock* command will return SUCCESS and the tuple will be unlocked so that other processes can access it. Table 4.5 summarizes the return codes of the *unlock* operation and their semantics.

Table 4.5 – Return codes of the *unlock* operation and their semantics.

Return Code	Semantics
SUCCESS	The process has successfully unlocked the tuple.
REMOVED	The tuple is not in TS either because the programmer neglected to return it to TS prior to attempting the *unlock* operation or an invalid *uid* value was specified.

After locking the tuple, the programmer will presumably remove the tuple from TS using the *inp* operation. At this point, he or she can verify that the data have not changed since the tuple was read and the decision was made to update the tuple. As a convenience to the programmer, however, the successful *lock* operation returns the data through its formal parameters. This allows the programmer to verify that the data have not changed since the decision was made to lock the tuple. If the data have changed and the programmer no longer wishes to update the tuple, then he or she can simply unlock the tuple and proceed. If the *lock* operation did not return the data, the programmer would have to remove the tuple from TS before verifying the data. In this situation, if the data were to change, then the programmer would have to execute an *out* operation to return the tuple to TS prior to unlocking it.

4.9 – The Revised Cooperative Protocol

With the addition of the *lock* and *unlock* operations, we revise the cooperative protocol to accommodate processes traversing the list.

4.9.1 – Inserting a node

To insert a node, the process traverses the list, reading each node, until the insertion point is located. For our purposes, the insertion point is the node after which the new node will be inserted. The process then locks the insertion point node using the *lock* instruction. Next, the process checks the return value of the *lock* operation to determine whether the lock was granted. If the lock is successful, the process then compares the data and pointers returned by the *lock* operation to the data and pointers returned by the previous *rdp* operation. This is done to determine if the insertion point node has changed during the elapsed time between reading and locking. If the node is still a valid insertion point, according to the programmer's criteria, the process then removes the insertion point tuple from TS, updates the pointers, places both the updated insertion point tuple and the new tuple into TS, and finally, unlocks the insertion point tuple.

The critical aspects of the insertion protocol are:

1) Locking the insertion point node; and
2) Checking to make sure the data and links have not changed between the time that the tuple was read and the time that the tuple was locked.

Locking the insertion point tuple is critical in order to signal, to other processes that may be traversing the list, that the missing tuple will be returning to TS. The data and links must be checked after locking the tuple in order to preserve the integrity of the LDS. To demonstrate the cooperative insertion protocol more clearly, we begin with an LDS as shown in 4.19, above.

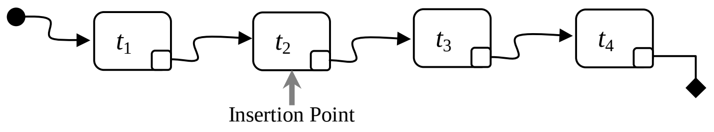


Figure 4.21 – Insertion point for new node.

Step 1 – read to the insertion point. Figure 4.21 illustrates the insertion point. The new tuple will be inserted after t_2.

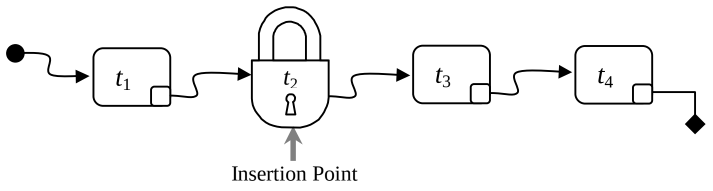


Figure 4.22 – Example list with the insertion point tuple locked.

Step 2 – Lock the insertion point tuple. Figure 4.22 shows the locked insertion point tuple. If the lock succeeds, the *lock* instruction returns SUCCESS. In addition to the return code, the *lock* instruction will return the data and pointer fields of the tuple, via its formal parameters, thereby allowing the programmer to verify that the tuple is still a satisfactory insertion point.

Step 3 – Remove the insertion point tuple from TS. Once insertion point has been verified as valid, the insertion point tuple is removed from TS using the *inp* operation. Figure 4.23 shows the state of the list in TS after the targeted insertion point is removed from TS. The dashed lines of t_2 represent the fact that, although the tuple is removed from TS, a trace still remains, in the form of the lock, so that the TSM can generate the appropriate failure code for the *rdp* or *inp* operation.

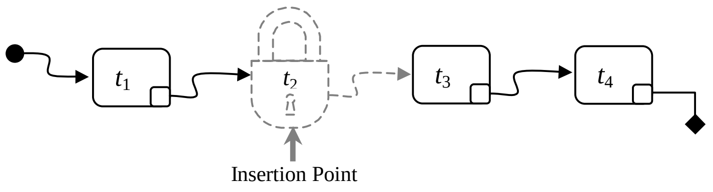


Figure 4.23 – Example list with the locked insertion point tuple removed from TS.

Step 4 – Store the new tuple in TS. Once the lock is granted, the data verified, and the insertion point tuple removed from TS, the new tuple is constructed. The new tuple is linked into the list by setting its ‘next’ pointer to refer to the tuple to which the ‘next’ pointer of the insertion point tuple currently refers. As complex as that sounds, it is actually quite simple, as illustrated by Figure 4.24, below. The new tuple is then stored in TS using the *out* operation. The state of the LDS, in TS, is reflected in Figure 4.24.

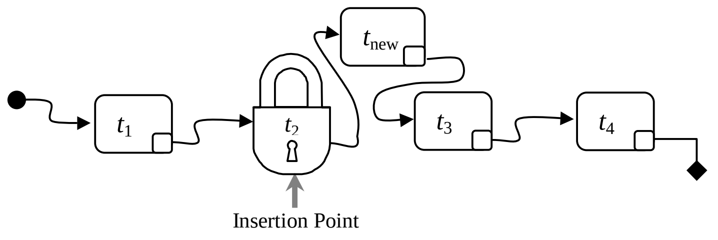


Figure 4.24 – Example list with the new node, t_{new}, inserted into TS.

Step 5 – Return the insertion point tuple to TS. Now that the new tuple representing the new node has been stored in TS, the insertion point tuple is returned to TS. The ‘next’ pointer of the insertion point tuple is set to refer to the new node that has been placed in TS. Figure 4.25 reflects the state of the LDS, in TS, after this step is complete.

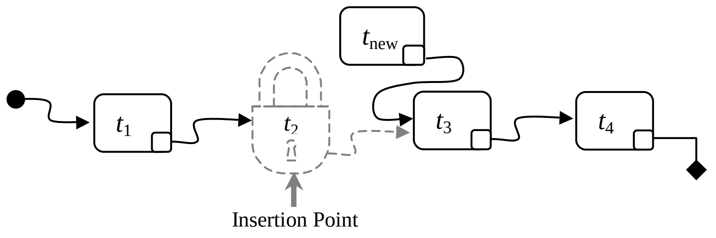


Figure 4.25 – Example list insertion completed (insertion point still locked).

Step 6 – Unlock the insertion point tuple. The final step in the insertion process is to unlock the insertion point tuple using the *unlock* operation. After the tuple is unlocked, the insertion is complete, and normal traversal of the list can resume. Figure 4.26 shows the LDS in its final state following the completed insertion of tuple t_{new}.

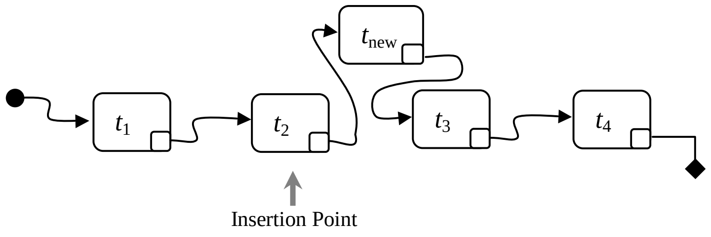


Figure 4.26 – Example list insertion completed (insertion point unlocked).

4.9.2 – Deleting a node

To delete a node from an LDS, the process must first determine the deletion point. The deletion point is the node immediately preceding the node to be deleted. The actual node to be deleted is referred to as the deletion target. Once the process finds the deletion

point node, a lock is placed on that node using the *lock* operation. If the *lock* returns SUCCESS, then the data returned by the operation are compared to those returned by the read operation. If the tuple still constitutes a suitable deletion point, then the deletion target is removed from TS using the *inp* operation. At this point, because the deletion target is not locked, the process must check the deletion target to determine if it has changed since it was last read. If the node is still suitable to be deleted, its predecessor – the deletion point tuple, is removed from TS. Otherwise, the deletion target tuple is returned to TS using the *out* operation, and the deletion point tuple is unlocked. If both the deletion point tuple and the deletion target tuple remain satisfactory, according to the criteria of the programmer, the *next* pointer of the deletion point node is updated to refer to the successor to the deletion target. The deletion point node is then returned to TS using the *out* operation and, subsequently, unlocked using the *unlock* operation.

The critical aspects of the deletion protocol are:

1) Locking the deletion point tuple,
2) Leaving the deletion target tuple unlocked,
3) Verifying that the deletion point tuple remains unchanged between the time it was read and the time it was locked, and

4) Verifying that the deletion target is remains unchanged between the time it was read and the time it was removed from TS.

The deletion point tuple must be locked to notify processes traversing the list that the tuple will be returned to TS. The deletion target tuple remains unlocked so that, once it is removed from TS, processes traversing the list will know that the tuple is not returning to TS. Finally, the data of both the deletion point tuple and the deletion target tuple must be examined to determine if any changes has been made that effect the decision to delete the targeted tuple. Additionally, the *next pointer* of the deletion point tuple must be examined. If this pointer has been changed then the list structure has changed since the deletion point tuple was read. To clarify the deletion protocol we will work through an illustrated example beginning with the initial state of the list in TS as depicted by Figure 4.19.

Step 1 – Traverse the list until the deletion point and deletion target are determined. Figure 4.27 illustrates an example list for which a deletion point and a deletion target have been identified. The deletion target will be removed from the list and the deletion point will have its pointer updated to bypass the deleted node.

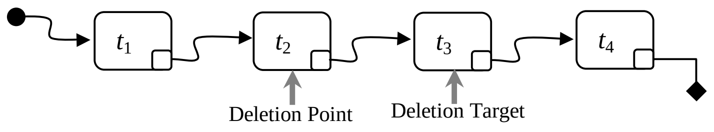


Figure 4.27 – Deletion point and deletion target are determined.

Step 2 – Lock the deletion point tuple. Figure 4.28 shows the locked deletion point node. The node will be removed from TS to have its pointer updated. The lock indicates to other processes that the tuple will return to TS. If the *lock* operation returns SUCCESS, then the data and pointer information returned by the *lock* operation are compared to the results of the last read operation to determine if changes to the deletion point tuple have invalidated the deletion target.

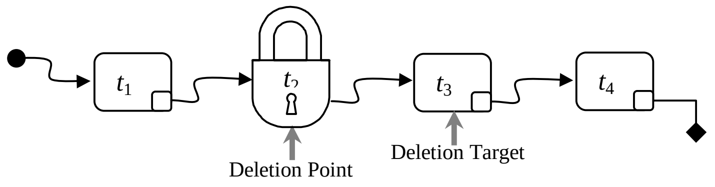


Figure 4.28 – Example list with deletion point node locked.

Step 3 – Remove the deletion point tuple and deletion target from TS. If the *lock* results indicate that the deletion target is still valid then the deletion target tuple is removed from TS. Note that the deletion target tuple is not locked prior to its removal from TS. The tuple remains unlocked because the expressed intention is to not return it to TS.

The possibility exists that the deletion target could have been changed during the elapsed time since it was identified as the deletion target and the time it was removed from TS. Therefore, it is necessary to check the contents of the deletion target tuple to verify that it is still the correct deletion target. If the deletion target tuple remains satisfactory, according to the programmer's criteria, then the deletion point tuple is removed from TS. Otherwise, the deletion target tuple must be returned to TS, using the *out* operation, and the deletion point tuple must be unlocked. Figure 4.29 illustrates the state of the example list after the deletion point tuple and the deletion target tuple have been removed from TS.

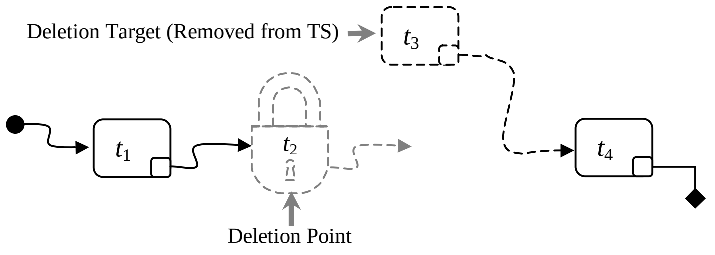


Figure 4.29 – Example list with locked tuple and deletion target removed from TS.

Step 4 – Update the pointers to bypass the deletion target. Figure 4.30 shows the 'next' pointer of the deletion point tuple updated to refer to the same tuple as the 'next' pointer of the deletion target.

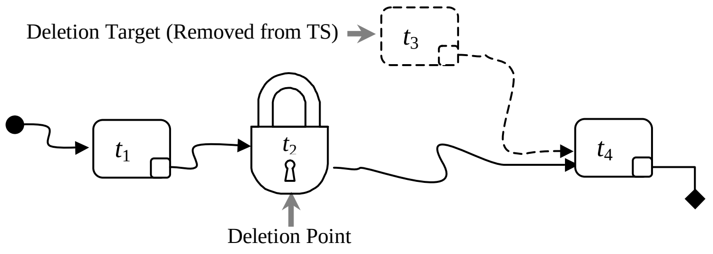


Figure 4.30 – Locked deletion point node with pointer updated to bypass deletion target.

Step 5 – Return the deletion point tuple to TS. After the 'next' pointer of the deletion point tuple is updated to bypass the deletion target, the deletion point tuple is returned to TS. Figure 4.31 shows the locked deletion point tuple after it is returned to TS with its pointer updated to bypass the deletion target.

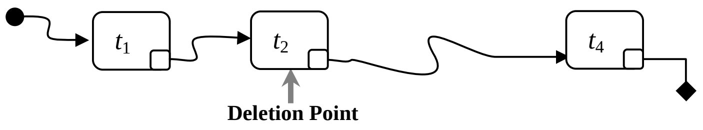


Figure 4.31 – Locked deletion point node returned to TS with updated pointer.

Step 6 – Unlock the deletion point tuple. The deletion process is completed by unlocking the deletion point tuple. When the deletion point tuple is unlocked, normal traversal of the list can resume. Figure 4.32 shows the example list after the deletion point tuple is unlocked.

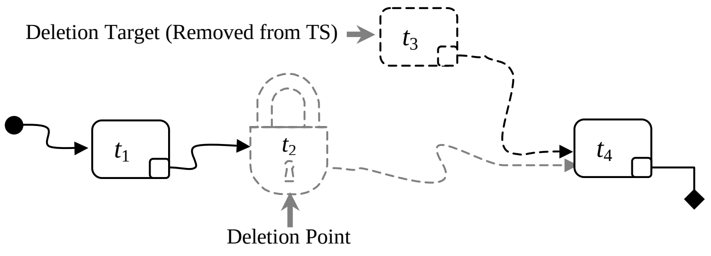


Figure 4.32 – Example list with deletion point unlocked and deletion complete.

4.9.3 – Updating a node

The cooperative update protocol is straightforward because it only involves one tuple – the update target. To update a node in the LDS, the process begins by locking the tuple to indicate to other processes that the tuple will be returning to TS. The locked tuple is then removed from TS; a check is performed to verify that the update is still necessary (i.e., to make sure no changes, since the last read, have obviated the need for an update); the changes are made; and, finally, the tuple is returned to TS and unlocked.

The critical aspects of the update protocol are:

1) Locking the update target tuple, and
2) Verifying that changes to the target tuple since the decision was made to update, if any, have not negated the need to update the tuple.

A detailed view of the cooperative update protocol begins with the initial state of the LDS depicted in Figure 4.19.

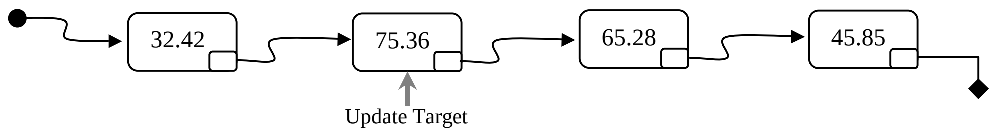


Figure 4.33 – Example list with the update target determined.

Step 1 – Identify the update target. A process begins the update procedure by traversing the list until the update target tuple is located. Figure 4.33 shows the example list with the update target tuple identified. The goal of the example update is to increment the datum in the target tuple by one.

Step 2 – Lock the update target. Once the update target is identified, according to whatever criteria the programmer is using, the next step in the cooperative update protocol is to lock the update target tuple using the *lock* operation. If the *lock* operation returns SUCCESS, then the data returned via the formal parameters of the *lock* operation are compared to the results of the *rdp* operation that was used to identify the update target tuple. Figure 4.34 illustrates the example list with the update target tuple locked.

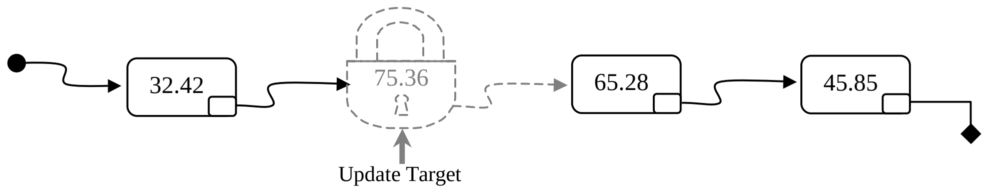


Figure 4.34 – Example list with the update target locked.

Step 3 – Remove the target tuple from TS. After the target tuple has been locked by the updating process, the next step is to remove the tuple from TS so that changes can be made to it. Figure 4.35 illustrates the example list with the locked update target tuple removed from TS.

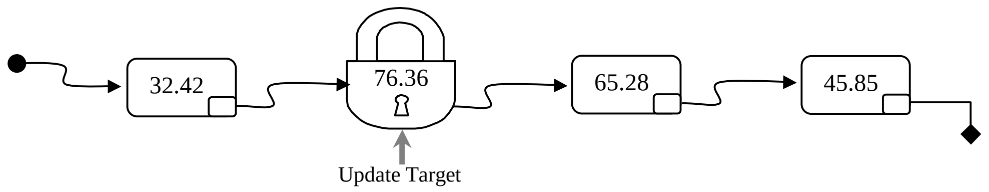


Figure 4.35 – Example list with the locked update target tuple removed from TS.

Step 4 – Return the updated tuple to TS. Once the process has exclusive access to the tuple in its local address space, it can update the data. In the current example, the updating process increments the value of the data element, by one, to 76.36. The tuple is then returned to TS using the *out* operation as shown in Figure 4.36.

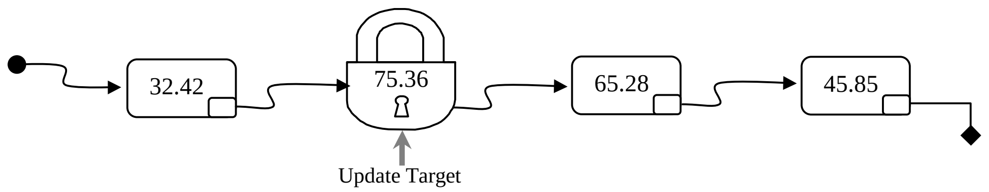


Figure 4.36 – Example list with the updated tuple returned to TS still locked.

Step 5 – Unlock the update target tuple. The final step in the cooperative update protocol is to unlock the update target tuple to make it accessible to other processes. Figure 4.37 shows the state of the example LDS after the update procedure is complete.

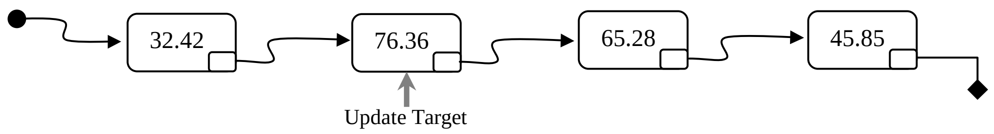


Figure 4.37 – Example list with the updated tuple unlocked and the update completed.

4.10 – Traversing the LDS under the Revised Cooperative Protocol

With the revised cooperative protocol, described in Section 4.9, in place we now examine the interaction between the new protocol and LDS traversal. The problem for the traversing process, in the absence of the revisions to the cooperative protocol, was that the process has no way of discriminating between a tuple that has been permanently removed from TS and one that has been temporarily removed from TS for modification. The changes to the cooperative protocol allow the process that is removing the tuple from TS to clearly indicate its intentions with respect to missing tuple. The following scenarios illustrate the ways in which the traversing process can employ the additional information to make better recovery decisions upon encountering a missing tuple.

The first scenario begins with the actual state of the LDS as shown in Figure 4.30. The deletion point tuple, t_2, has been locked and then removed from TS by another process, P_1, for updating. Process P_2 is traversing the list, has just read t_1, and is preparing to follow the link from t_1 to t_2. Process P_2 executes a *rdp* operation to read tuple t_2. The operation fails with the return code LOCKED. At this point, P_2 begins to execute a hard wait, looping until t_2 is unlocked by process P_1 at which time the *rdp* operation succeeds and returns the data to P_2. Figure 4.32 illustrates the state of the LDS in TS at this point.

The second scenario is a bit more complicated. In this case, process P_2 picks up the reference to the next node in the list before that node is deleted. Table 4.6 shows the timeline of events involved in process P_2 traversing the LDS while process P_1 is actively deleting a node from the same range of nodes within the LDS.

Table 4.6 – Timeline of events involved in a concurrent traversal of and deletion from an LDS in TS

Time Θ_n	Process P_1	Process P_2
Θ_{01}	rdp(t_2)	rdp(t_2)
Θ_{02}	lock(t_2)	–
Θ_{03}	inp(t_3)	–
Θ_{04}	inp(t_2)	rdp(t_3) – returns REMOVED
Θ_{05}	out(t_2)	–
Θ_{06}	–	rdp(t_2) – returns LOCKED
Θ_{07}	–	Loop waiting for *rdp*(t_2) to return SUCCESS
Θ_{08}	Unlock(t_2)	–
Θ_{09}	–	rdp(t_2) – returns SUCCESS
Θ_{10}	–	rdp(t_4) – returns SUCCESS

Figure 4.38 shows the state of the LDS in TS, relative to the activities of processes P_1 and P_2, at time Θ_1. At time Θ_1, both P_1 and P_2 have read tuple t_2. Process P_1 has identified tuple t_2 as the deletion point and tuple t_3 as the target of the deletion. Process P_2, on the other hand is simply traversing the LDS. As such, P_2 has identified t_3 as the next node to be read in the LDS.

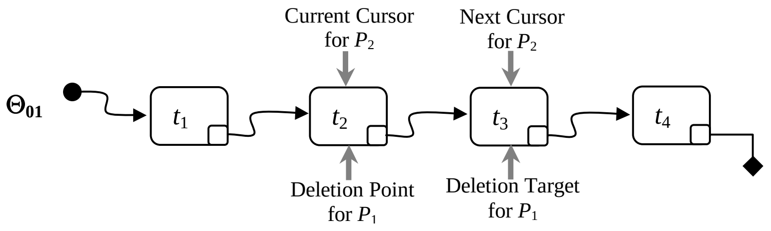


Figure 4.38 – Deletion point and deletion target are identified by P_1.

However, before P_2 can follow the link from t_2 to t_3, process P_1 locks t_2, then removes t_3 from TS, and finally removes t_2 from TS. Therefore, as Table 4.6 shows, when P_2 executes the *rdp* operation on t_3, the operation returns the REMOVED code. Figure 4.39 illustrates the state of the LDS in TS at time Θ_{04}, at which point, P_2 attempts the *rdp* operation on t_3.

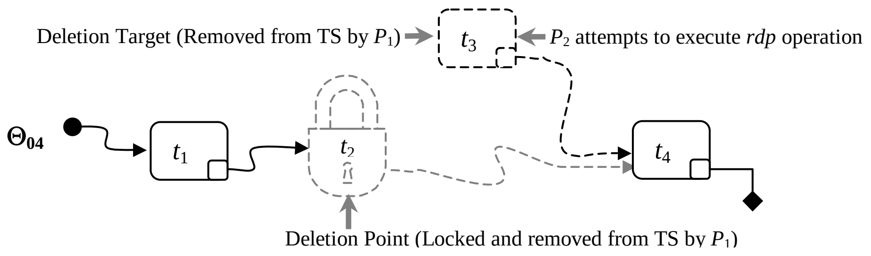

Deletion Target (Removed from TS by P_1)
t_3
P_2 attempts to execute *rdp* operation
Θ_{04}
t_1
t_2
t_4
Deletion Point (Locked and removed from TS by P_1)

Figure 4.39 – Locked deletion point node with deletion target removed from TS.

The REMOVED code indicates to P_2 that tuple t_3 is not returning to TS. At this point, the only recourse P_2 has is to back up to the preceding tuple and pickup the new link. To accomplish this, P_2 executes a *rdp* operation on t_2 at time Θ_{06}. Immediately prior to that point in time, P_1 returns t_2 to TS (at time Θ_{05}) but it remains locked as shown in Figure 4.40.

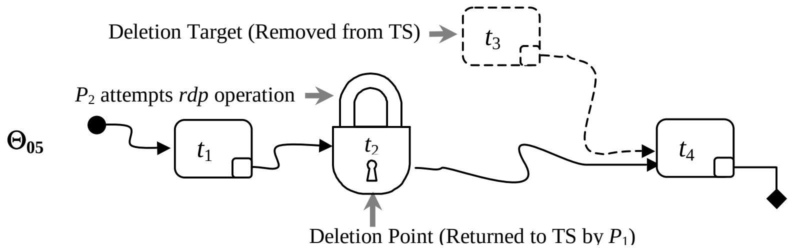

Deletion Target (Removed from TS)
t_3
P_2 attempts *rdp* operation
Θ_{05}
t_1
t_2
t_4
Deletion Point (Returned to TS by P_1)

Figure 4.40 – Locked deletion point node returned to TS with updated pointer.

Therefore, when P_2 attempts the *rdp* on t_2 at time Θ_{06}, the operation returns the LOCKED code. The LOCKED code indicates to P_2 that the tuple will be returned to TS at some time in the future. The course of action taken by P_2, in this example, is to enter a hard wait, looping until t_2 is returned to TS.

The scenario continues with P_1 unlocking t_2 at time Θ_{08}. Subsequently, the hard wait undertaken by P_2 executes another *rdp* operation at time Θ_{09}. Figure 4.41 shows the state of the LDS in TS at that time.

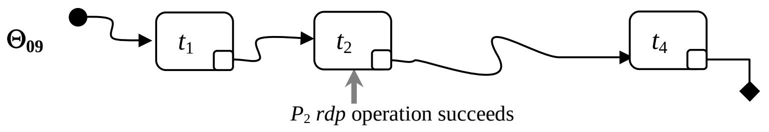


Figure 4.41 – Example list with deletion point unlocked and deletion complete.

With t_2 unlocked, the *rdp* operation at Θ_{09} succeeds and P_2 is able to continue traversing the list. By backtracking and reading t_2 after the attempt to read t_3 failed, P_2 picks up the updated reference to t_2's new successor. At time Θ_{10}, P_2 successfully reads tuple t_4 and is back on track.

4.11 – The contract of the lock operation guarantees that a locked tuple will be returned to TS. Given that fact, can a reading process, upon encountering a locked tuple, issue a non-predicate rd operation on that tuple without fear of indefinite blocking?

While it is true that a locked tuple will be returned to TS sometime in the future, there is still plenty of room for problems. It is entirely possible for the tuple to be returned to TS, unlocked, and another process to permanently remove the tuple from TS between the time the reading process identifies that the tuple is locked and the time that it executes the non-predicate *rd* operation. If that were to happen then, when the reading process executes the *rd* operation, it will block indefinitely.

4.12 – Why can we not ensure the success of a non-predicate rd operation by allowing it to execute as soon as a locked tuple is unlocked, thereby eliminating the need for a reading process to execute a "hard wait?"?

We can certainly do this for the non-predicate *rd* operation. However, doing so leads to bizarre inconsistencies between the use of the non-predicate *rd* operation and the non-predicate *in* operation. It is impossible to achieve the same guarantee for the *in* operation because if we queue an *in* operation for execution immediately upon unlocking the tuple, then any subsequently queued non-predicate operations are subject to indefinite blocking if the tuple is not returned to TS. Therefore we are faced with the choice between forcing a reading process to execute a "hard wait" and allowing the use of one non-predicate operation – the *rd* operation – while disallowing the use of the *in* operation.

Chapter 5 – Pointer Support for GC: Level 3

This chapter examines the construction and manipulation of LDS in TS within an environment in which processes concurrently accessing nodes of the LDS compete against one another.

This chapter addresses the following issues:

- Is it sufficient, in the face of competition, to implement operations that follow the cooperative protocol so that competing processes cannot bypass any critical steps?
- Given that we cannot, by definition, trust competing processes to make altruistic decisions with respect to the integrity of the LDS, how do we support competitive concurrency?
- How can the TSM make decisions on behalf of the programmer without knowing his or her intentions?
- What is the proper balance between parsimony and flexibility for the programmer?
- What low-level support must be incorporated into the TSM in order to allow the Level 3 operations to be successfully implemented?
- What changes to traditional Linda syntax are required to accommodate the Level 3 LDS operations?
- What are the semantics of each of the Level 3 LDS operations?

5.1 – Overview

The exploration of Level 1 illustrates that the *uid* data type is both necessary and sufficient to support sequential access to an LDS in TS. However, it is insufficient to support concurrent access to the LDS without compromising the structural integrity and the data integrity of the LDS. Level 2 facilitates concurrent access via the addition of a cooperative protocol that must be followed by processes accessing an LDS in TS. Additionally, Level 2 provides feedback to reader processes by requiring, as part of the cooperative protocol, processes to use the *lock* operation to signal their intentions with respect to LDS tuples that they remove from TS.

The key to successful concurrent access to the LDS at Level 2 is the cooperation between processes. Where practicable, Level 2 functionality provides concurrent access to an LDS with minimal additional overhead relative to sequential access. However, there exist situations in which it cooperation is either impossible or impractical for any number of reasons. In the absence of cooperation between processes we have reverted, essentially, back to Level 1 functionality. The problems associated with concurrency at Level 1 are well documented in the motivation for Level 2 (See Section 4.3). This chapter, therefore, develops solutions for the problems that arise in an environment in which the processes are competing for concurrent access to the nodes of an LDS in TS.

5.2 – Is it sufficient, in the face of competition, to implement operations that follow the cooperative protocol so that competing processes cannot bypass any critical steps?

At a glance, it appears that the most straightforward solution to the problems of competitive concurrent access is to introduce operations that enforce the Level 2 cooperative protocol. To implement this solution we develop three high-level operations: insert, delete, and update. The exact parameter signature of each operation is fully developed along with a viable solution. However, there are a few obvious requirements that suffice for discussion purposes. First, there is no doubt that the insert operation needs to be provided with the tuple to be inserted into the list. It also requires the insertion point tuple (i.e., the tuple after which the new node is inserted). Similarly, the *delete* operation requires the target node (the node to be deleted) and the *deletion point* tuple (the target's immediate predecessor). Finally, the update operation would, at a minimum, require a tuple that matches the update target in terms of its parameter signature, with the desired changes to the actual parameter values.

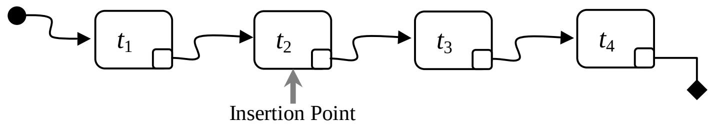


Figure 5.1 – Example LDS with insertion point tuple identified.

However, as the following scenario illustrates, the obvious solution is not possible. The scenario describes the process of inserting a new node. It begins by traversing the list until the insertion point is identified. Figure 5.1 shows the list with the insertion point identified. As in Level 2, the insertion point refers to the node after which the new node is to be added.

After the insertion point is identified, the Level 3 *insert* operation is called. If we adopt the seemingly obvious solution to the problems of competition, the *insert* operation will begin to execute the steps of the Level 2 cooperative protocol. The first step of the Level 2 cooperative protocol requires the insertion point tuple to be locked.

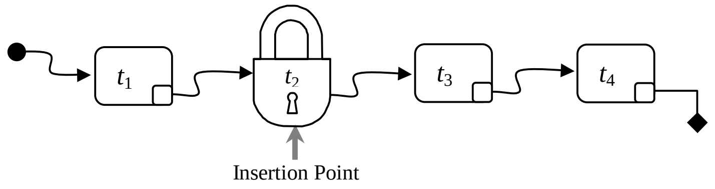


Figure 5.2 – Example list with the insertion point tuple locked.

Figure 5.2 shows the list with the insertion point tuple locked. This is the point after which the obvious solution hits a brick wall. A successful call to the *lock* operation returns the contents of the locked tuple. The central tenet of the cooperative protocol is that no action can continue if the process does not have an accurate view of the current state of the LDS in TS. The process is responsible for comparing the actual state of the LDS in TS, as returned by the *lock* operation, to the perceived state of the LDS. If the node has changed since the process read it, then the process must, per the cooperative protocol, abort the operation. It is this aspect of the cooperative protocol that makes the naïve approach to competitive concurrency untenable. It is not possible to leave it up to the competitive process to verify that the tuple, that represents the target node of the LDS, has not changed since the decision was made to insert the new node after the insertion point. Furthermore, if the insertion point tuple was changed before it could be locked, the programmer must decide how to proceed. This step of the cooperative protocol is totally incompatible with the competitive environment for a very simple reason – the locus of the decision resides at too low a level for programmer-level code to make the decision. The decision point exists, and must be addressed, at the system-level code (i.e., within the *insert* operation itself). The locus of control must reside in the system-level code because, in the competitive environment, the programmer cannot be ‘trusted’ to play nice. It is a serious threat to the structural integrity of the LDS to allow circumstances in which a programmer could say: “I don’t care if the tuple has changed – I’m always going to proceed with my changes regardless of the current state of the tuple.”

5.3 – Given that we cannot, by definition, trust competing processes to make altruistic decisions with respect to the integrity of the LDS, how do we support competitive concurrency?

Having established that the locus of control must be at the system-level, a new issue arises. When manipulating a linked list a variety of decision criteria can be used to determine where a modification is to occur. For example, when inserting a node, the decision of where to effect the insertion might be based upon the value of the node preceding the newly inserted node. Alternatively, the decision might be based upon the value of the node immediately succeeding the newly inserted node. The decision to insert a new node in a particular place might be based upon both the immediate predecessor and the immediate successor of the new node. Or, it might be the case that the value of the nodes is irrelevant and that the decision to insert is based upon the position, relative to the head of the list, of the newly inserted node. Regardless of the decision criteria used, at Level 2, because the decision of whether or not to proceed with an action is left at the programmer-level, the programmer can incorporate code to verify whether or not any changes have occurred, between the time the decision was made to modify the list and the time at which the relevant nodes are reserved for exclusive use. If changes have occurred, the programmer can decide whether or not those changes have rendered the decision invalid according to his or her criteria.

For example, assume that a program is building an ordered linked list. The decision is made to insert the node with a key value of 14 between the two nodes with key values of 10 and 17, respectively. Further assume that, before the requisite nodes can be locked, that the key value of the predecessor changes from 10 to 12. At Level 2, the programmer-level code can make the determination that the change does not affect the desired end result and therefore the code can proceed to modify the list. If, on the other hand, the value of the preceding node had changed from 10 to 15, then the proposed insertion of the value 14 would lead to an invalid result.

At Level 3, however, the decision of whether to proceed with the modification or not, must be made at the system-level. Therefore, it becomes impossible for decisions to be made based upon the intent of the programmer. The decision criteria must be generic in nature.

5.4 – How can the TSM make decisions on behalf of the programmer without knowing his or her intentions?

For any given LDS operation, excluding the *read* operation, there exists a *critical range* (see Figure 5.3) consisting of the target node, the predecessor of the target node and the successor to the target node.

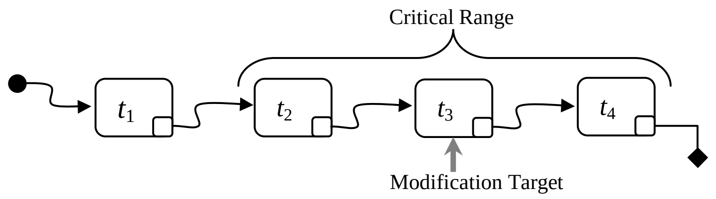


Figure 5.3 – Example LDS with modification target and critical range identified

The issue, at this point, is deciding when an operation should fail, given that one or more of the nodes in the critical range has changed. One line of reasoning would argue that separate operations should be provided so that the programmer could specify, by using the appropriate operation, whether a particular change would be ignored. For example, the Level 3 library would provide, perhaps, an *InsertAfter* operation that would fail if the predecessor to an insertion operation were altered between the time the decision was made to insert and the time that the predecessor was locked for exclusive access. The *InsertAfter* operation, however, would ignore changes to the proposed successor to the new node. This logic leads one to postulate the need for an *InsertBetween* operation that would fail if either the predecessor or successor were to change and an *InsertBefore* operation that would fail if the successor were to change.

5.5 – What is the proper balance between parsimony and flexibility for the programmer?

These operations seem reasonable within the context of insertion of new nodes into the list. However, the same logic leads to some bizarre outcomes when the context switches from insertion to updating or deleting a node. We are now faced with the need for a *DeleteAfter* operation that will ignore changes to the successor of a deletion target. Additionally, the attempt to provide maximal flexibility to the programmer begat the need for a rather strange *DeleteBefore* operation that ignores changes to the predecessor of a deletion target but fails if the successor changes. We are also led, following this line of reasoning to its logical conclusion, to the need for *DeleteBetween*, *UpdateAfter*, *UpdateBefore*, and *UpdateBetween* operations.

While some valid arguments can be made for including this level of control in the Level 3 operations, in the interest of parsimony, we adopt the position that the Level 3 *Insert*, *Update*, and *Delete* operations will fail if any of the nodes in the critical range have been modified, regardless of whether the modification is to data or to pointers.

In addition to keeping things simple, the critical range principle lets us escape from a paradox caused by allowing the Level 3 operations to ignore changes to the predecessor of the proposed modification target. Consider the following situation that could arise were we to implement the *DeleteBefore* operation. The *DeleteBefore* operation fails if the successor to the deletion target changes, but ignores changes to the predecessor of the deletion target. Figure 5.4 illustrates the state of the example LDS at the point in time at which the decision to delete is made.

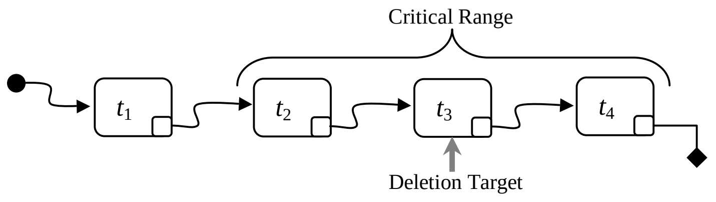


Figure 5.4 – Example LDS with deletion target identified.

After identifying the deletion point, the next step is to call the *DeleteBefore* operation. Between the time that the decision was made to delete tuple t_3 and the time that the *DeleteBefore* operation is called, another process deleted tuple t_2, which was previously in the critical range for the deletion operation. Figure 5.5 shows the actual state of the LDS at the time the *DeleteBefore* operation was called.

The predecessor to the deletion target has changed. The *DeleteBefore* operation is supposed to ignore the change. However it is clear, given the process of deleting a node from any linked list, that the change cannot be ignored in reality. In order to delete a node, the predecessor of that node must be updated to point to the successor of that node.

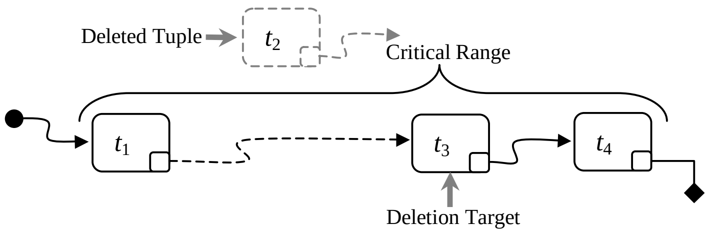


Figure 5.5 – Example LDS with target predecessor link modified.

Therefore, in order to make the *DeleteBefore* operation appear to ignore the change to the predecessor, from the programmer's perspective, the TSM must maintain backwards links to the predecessor of each node. The paradox is, thus, that a doubly linked list must be created in order to successfully manage the *DeleteBefore* operation on a singly linked list. The paradox is avoided by making a straightforward, orthogonal set of instructions that take the same three parameters – the predecessor node, the target node, and the successor node.

5.6 – What low-level support must be incorporated into the TSM in order to allow the Level 3 operations to be successfully implemented?

This section illuminates the need for low-level support operations in the TSM that are not necessary at Level 1 or at Level 2. Following the motivation, we discuss each operation in detail. The low-level operations fall into two classifications: public and private. Public operations are those that will be directly called by programmer-level code. Private operations, on the other hand, are only called by the system-level operations at Level 3. The public operations are the *connect* operation and the *disconnect* operation. The private operations are the *modified* operation, the *reserve* operation and the *release* operation.

5.6.1 – The *connect* Operation

With the nascence of competitive concurrency at Level 3, arrives the need to uniquely identify processes making requests of the TSM. At Level 1, only one process accesses the list at any given time. Therefore, no need exists to uniquely identify the process. At Level 2, concurrent processes cooperate to ensure that the structural integrity and the data integrity of the LDS remain intact. Each processes makes its own decision, as to whether or not to proceed with the proposed operation, based solely upon information in its local address space. Therefore the identity of the process is irrelevant.

At Level 3, however, the TSM must make decisions for the processes to prevent threats to the structural integrity and the data integrity of the LDS. For this to occur, the TSM must be able to uniquely identify each process so as to be able to piece together a serialized picture of the actions taken against the LDS by each process. To better understand the need for this functionality, consider the following scenario. Two processes are operating simultaneously on an LDS in TS. Process P_1 will insert a new node into the LDS. Process P_2 will delete a node from the LDS.

Table 5.1 – Timeline of events associated with a competitive, simultaneous update of an LDS.

Time Θ_n	Process P_1	Process P_2
Θ_{01}	read(t_2)	–
Θ_{02}	Decide to insert after t_2	read(t_2)
Θ_{03}	Construct t_{new}	read(t_3)
Θ_{04}	read(t_3)	read(t_4)
Θ_{05}	–	Decide to delete t_3
Θ_{06}	–	delete(t_2, t_3, t_4), succeeds

Θ_{07}	insert(t_2, t_{new}, t_3), fails	–

Table 5.1 shows the sequence of events involved in the scenario at hand. Figure 5.6 illustrates the initial state of the LDS in TS as the action begins.

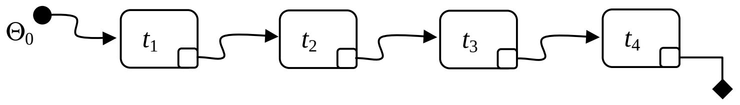


Figure 5.6 – Initial state of LDS (time Θ_0).

Process P_1 reads through to tuple t_2. At time Θ_2, P_1 makes the decision to insert a new node after tuple t_2 and constructs the new tuple, t_{new}, at time Θ_3. At time Θ_4, P_1 reads the successor to t_2 in preparation to call the *insert* operation. However, before the *insert* operation can be called, a context switch occurs and P_1 is blocked. In the mean time, P_2 has read the tuples comprising the critical range to delete tuple t_3. At time Θ_6, P_2 successfully deletes tuple t_3, leaving the LDS in the state illustrated by Figure 5.7.

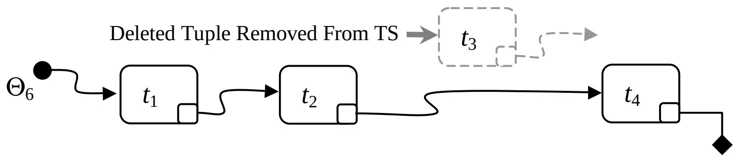


Figure 5.7 – Actual state of LDS following deletion of t_3 by P_2 (time Θ_6).

The scenario continues as P_1 resumes where it left off. At time Θ_7, the process attempts to call the *insert* operation passing the members of the relevant critical range, as it perceives them to be.

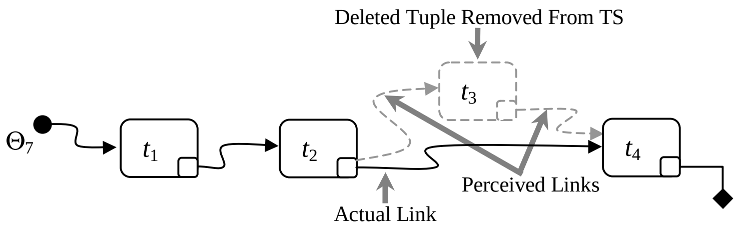


Figure 5.8 – State of LDS as perceived by P_1 prior to insertion (time Θ_7).

Figure 5.8 reflects the *inconsistent view*, held by process P_1, of the state of the LDS in TS. It is clear that, if the new node is inserted by P_1 after t_2 without accounting for the changes made by P_2, then the structural integrity of the LDS will be compromised. At Level 3, the responsibility for preventing the *insert* operation from completing rests with the TSM. However, the TSM cannot determine the sequence of operations performed by the various processes unless it can uniquely identify each of those processes.

The naïve approach to uniquely identifying processes would be to have the process retrieve its unique local process id from the operating system on its host machine and pass that along with the operations it performs on an LDS in TS. However, this approach is prone to failure because the processes might be running on different machines. This makes the system susceptible to coincidental duplicate process ids from each of the various machines on which the processes are executing.

Another, more solid approach, might be to have the processes pass a unique process identification constructed by concatenating the machine IP address with the local process identifier. Even this, however, is open to potential problems. Many operating systems reset the process id counter when the machine is rebooted or when the maximum process id value has been used. It is, therefore, possible that a duplicate process id could be constructed even with the addition of the IP address to the identifier.

Instead, we opt to shift control of uniquely identifying processes to the TSM itself. Processes will be required to call the *connect* operation prior to executing any other Level 3 operations. The *connect* operation will establish a connection to the TSM and request, from the TSM, a unique TS Access key (TSAK) that will be used by the process on all subsequent calls to the TSM. If a process attempts to call a Level 3 operation without a duly assigned TSAK, then the TSM will raise an exception and the operation will be aborted.

5.6.2 – The *disconnect* operation

The motivation for the *disconnect* operation is very simple. Having established the need for the *connect* operation, common sense mandates the existence of a complementary operation to signal to the TSM that a particular process will no longer be making calls to the TSM. When a process calls the *disconnect* operation, the TSM can then recycle the TSAK. Additionally, records of operations on the LDS that are identified with the recycled TSAK can be purged.

Given that we are operating in a competitive environment, we cannot be assured that the *disconnect* operation will be called by all processes. The *disconnect* operation is a courtesy and convenience to the TSM – much like the *dispose* method of a Java class. In Java, when an object is no longer referenced, the garbage collection routine will reclaim the memory allocated for that object. However, the programmer can call the dispose method to force the memory to be returned to the heap. In this case, subsequent sections will detail the nature of the activity records maintained by the TSM to support Level 3 operations, and the natural limits on the number of entries that must be maintained for each node in a particular LDS. That discussion will demonstrate activity records indexed by inactive TSAK values will eventually be purged through normal attrition.

5.6.3 – The *modified* operation

In Section 5.6.1, considerable discussion was devoted to motivating the need for the TSM to uniquely identify each process making Level 3 calls to the TSM. The TSM needs to be able to uniquely identify each process so as to ensure that the operations on a given LDS are serializable. In other words, at Level 3, the TSM has the sole responsibility of guaranteeing that the state of an LDS, following any series of parallel operations on the LDS, is identical to what the state of the LDS would have been had those same operations been executed sequentially.

The key to successfully determining whether or not a set of operations is serializable is the following observation. If a particular node has not been modified since a given process last read it, then the process has a consistent view of the actual state of that node of the LDS as it exists in TS. The question, then, becomes: "How does the TSM know if a given process has read a particular node of the LDS since that node was last modified?" The most obvious answer to the question at hand is to keep a list, for each node of each LDS, of the activities of every process operating on that list. When a particular process attempts to execute a Level 3 operation, the *read* operation notwithstanding, that process will call the *modified* operation. The *modified* operation only needs to access the activity list for the nodes that comprise the critical range of the operation that the process is attempting to execute. If no other process has modified the nodes of the critical range since the current process has last read those nodes, then *modified* returns false and the operation can be carried out successfully. If, on the other hand, one of the nodes in the critical range has been modified since the current process last read it, then *modified* returns true and the proposed operation must fail. It might be the case that the operation could, in fact, be carried out successfully. However, there is no way to know whether or not that is the case. Therefore, if any of the nodes comprising the critical range have been modified since the process initiating the operation last read them, then the operation must, inexorably, fail in order to protect the integrity of the LDS.

5.6.4 – Limits on the growth of node access journals

Section 5.6.3 highlights the need for a *modified* operation to determine if a particular node, in the critical range of an *insert*, *update,* or *delete* operation, has been modified since the process that is attempting to call that operation last read it. The mechanism to make this determination consists of a journal of activities made against a particular node by all of the processes operating upon that node. The natural concern is that the length of the journal for each node will experience unbounded growth. Another issue derives from this concern. As the size of the journal increases for each node, so too, will the search time required to answer the *modified* operation. It is easy to envision a scenario in which, for a persistent list consisting of a large number of nodes, the response times to the *modified* operation slow down so much that the TSM has trouble responding to any other requests.

How, then, can the size of the journal be limited so as to prevent the unacceptable scenario, described above, from occurring? A critical first step to resolving this issue is the following observation. It does not matter whether a process has read a particular node once, or ten thousand times, since it was last modified. The process gains no additional knowledge about the state of the LDS in TS on the ten-thousandth post-modification read than it gained on the first post-modification read. This observation allows us to reduce the size of the journal by storing a record of only the first post-modification read a process makes on a particular node.

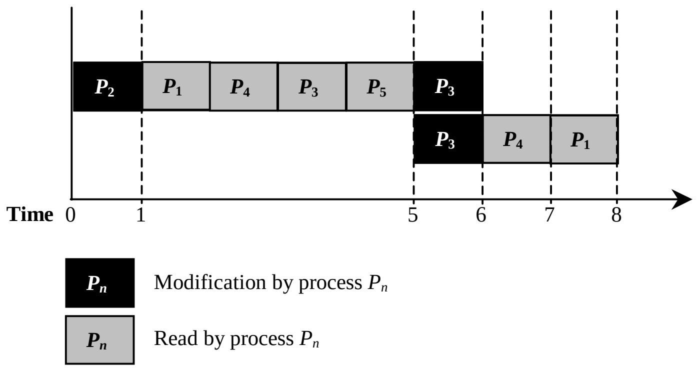


Figure 5.9 – Serialized timeline of accesses to a node of an LDS.

The next issue to arise is that of how long the journal entries must persist. As Figure 5.9 illustrates, this issue also finds its resolution in a basic observation. Assume that the initial modification on an LDS node in TS occurs at time 0 and ends at time 1. From time 1 through time 5, a total of four processes read the node. These four processes, along with process P_2, share a consistent view of the state of the node in question at the time P_2 completed its modification. At time 5, however, P_3 modifies the node again. At this time, only one process can possibly have an accurate view of the state of the node in TS – that is P_3. Process P_3 has a consistent view of the state of the node because it made the changes. Other processes must reread the node in order to be aware of the changes P_3 made to the node. The conclusion drawn from this observation is that, once a node has been modified, all previous actions taken on that node are irrelevant because they do not provide any information, whatsoever, regarding the current state of the node in TS.

Therefore, the TSM can purge any journal entries that occur prior to the most recent modification, thereby ensuring that the size of the journal for each node is minimal.

5.6.5 – The *reserve* and *release* operations

The *critical range principle* (see Section 5.4) mandates that the predecessor, successor, and the target (for *delete* and *update* operations) remain unchanged between the time that a decision is made to perform the operation and the time that the operation is completed. This fact indicates the need to ensure mutually exclusive access, by the modifying process, to the nodes in the critical range. It is certainly true that the modifying process could simply remove the nodes comprising the critical range from TS, thereby ensuring exclusive access to those nodes. However, so doing would lead to another instance of *the reader's problem* as discussed in Section 4.6. Simply put, the *reader's problem* occurs when a process is traversing an LDS in TS and encounters a node that has been removed from TS. How is the process to proceed? If the node has only been removed from TS temporarily, then the process can simply attempt to reread the node until the read operation succeeds. However, if the node has been removed from TS permanently, then the preceding strategy would lead to a dead end. Instead, if the node has been permanently removed from TS, then the reader process must back up to the previous node in the list and pick up the pointer to the new successor of that node.

At Level 2, the solution was to implement the *lock* operation (See Section 4.6 and Section 4.7) and incorporate it into the cooperative protocol. This extension to GC requires that a process call the *lock* operation, prior to removing an LDS tuple from TS, if that tuple will be returned to TS at some time in the future. If, on the other hand, the tuple will not be returned to TS, then the process simply issues an *inp* operation removing the tuple from TS without locking it. In this way, when a reader process encounters a tuple that is missing it can receive a differential failure code indicating which of the two situations exists. At a glance, it seems that the solution at Level 3 would simply be to call the Level 2 *lock* operation. However, that strategy is untenable for the following reasons. First, the Level 2 *lock* operation is a public operation, meaning that it can be called by programmer-level code as opposed to just system-level code. That fact, in and of itself, is not a major problem, however, the Level 2 *lock* operation is not sufficiently secure for the competitive environment of Level 3. Any process can unlock a tuple locked by the Level 2 *lock* operation. This is not a problem at Level 2 because processes at Level 2 are cooperating. No process, at Level 2, would maliciously unlock a tuple locked by another process. At Level 3, however, processes cannot be trusted not to maliciously unlock a tuple locked by another process.

Given that the Level 2 *lock* operation is insufficient at Level 3 to solve the same problem, Level 3 obviously requires an operation similar to the Level 2 *lock* operation. However, the Level 3 operation must require that the locking process provide the operation with a valid TSAK when locking the tuple. Furthermore, the Level 3 equivalent of the *unlock* operation must require a valid TSAK and only unlock those tuples that were locked using the same TSAK.

To accomplish the goals describe above we introduce the *reserve* operation. The Level 3 *reserve* operation behaves similarly to the Level 2 *lock* operation. As does the Level 2 *lock* operation, the *reserve* operation addresses the *reader's problem*. However, the *reserve* operation is a private operation, the use of which is restricted exclusively to system-level code. The *reserve* operation requires, in addition to the tuple to be reserved, the TSAK of the process requesting the reservation.

As the cliché states, what goes up must come down – in this case, what gets reserved must also get unreserved. Therefore, to undo the *reserve* operation we introduce the *release* operation. The *release* operation takes two parameters: the *primary uid* of the tuple to be released and the TSAK of the process that reserved the tuple for exclusive access. The *release* operation will fail if the TSAK provided by the process attempting the release does not match the TSAK provided by the process that executed the *reserve* operation.

5.7 – High-Level LDS Operations

The low-level support operations developed in Section 5.6 are the bricks with which we build the high-level LDS operations to construct and maintain a singly linked list within the context of a competitive, concurrent environment. This section lays those bricks in the proper sequence to ensure that the operations will withstand the winds of competitive concurrency without deleterious effects to the data integrity and the structural integrity of the LDS.

5.7.1 – What changes to traditional Linda syntax are required to accommodate the Level 3 LDS operations?

Traditional GC syntax requires the command name followed by the left parenthesis, followed by one or more parameters, either formal or actual, followed by the right parenthesis. The partial BNF grammar shown in Figure 5.10 illustrates this unambiguously.

```
  <command> →  <instruct> ( <paramList> ) ;
 <instruct> →  in | rd | inp | rdp | out | eval
<paramList> →  <param> , <paramList> | <param>
    <param> →  <formal> | <ident>
   <formal> →  ? <ident>
    <ident> →  Any legal C language identifier
  <literal> →  Any literal value legal in the C language
```

Figure 5.10 – Partial BNF grammar for traditional GC command syntax.

In the traditional GC command syntax, the parameters of the command are, themselves, fields of the tuple being stored or retrieved. The Level 3 commands require significantly more information. According to the *critical range* principle (Section Section 5.4), the Level 3 *insert*, *delete*, and *update* commands require the predecessor tuple, the target tuple, and the successor tuple. All of the Level 3 commands, including the *read* command, require the programmer to pass a valid TSAK (See Section 5.6.1) so that the TSM is able to track the sequence of operations performed on each node. Because of this fact, we must adopt a new syntactical convention for the all of the Level 3 commands. At issue here is how to list, as parameters to a given command, as many as three separate tuples and the TSAK.

The first step to defining the new syntactical convention for the Level 3 commands is to determine whether or not the predecessor tuple and the successor tuple must be fully specified (i.e., specifying all of the fields of the tuple). The alternative is to simply specify the *primary uid* of each of those tuples. Figure 5.11 illustrates the two alternatives.

```
insert((predUID,?intData2,targetUID),(^targetUID,targetData,succUID),
       (succUID,?intData,?succUID2));
```

insert operation with predecessor tuple and successor tuples fully specified.

```
insert(predUID,(^targetUID,targetData,succUID),succUID);
```

insert operation with predecessor tuple and successor tuples fully specified.

Figure 5.11 – Alternatives for passing parameters to the Level 3 operations.

To resolve this issue, we must examine the reason for fully specifying a tuple under general circumstances. In traditional GC, the purpose of fully specifying all of the fields of a tuple is twofold. First, in the case of storing the tuple, the fields of that tuple, clearly, must have actual values specified for each field so that the tuple can be stored. In the case of retrieval, the parameter signature of a tuple consists of the position of the parameter, its data type and the value of the parameter if it is an actual parameter. The parameter signature is used to find and retrieve the matching tuple from TS.

With the incorporation of pointer semantics into GC, we now have the ability to uniquely identify a tuple in TS solely on the basis of its *primary uid* value. So why, then, must we still fully specify the tuples in retrieval commands given that we can locate them using their *primary uid* values? The answer is straightforward and constrained by practical concerns. While it is true that we can identify the tuple for retrieval using its *primary uid*, we still need to have the parameters fully specified because the process calling the retrieval command must provide the addresses of storage allocated in its local address space for the data to be returned.

We have established that the reason for fully specifying *uid*-containing tuples is either to provide actual values in the case of storage commands, or to provide the address of storage space for returned values in the case of retrieval commands. Given this fact, we easily make the case that there is no need to fully specify the predecessor tuple and the successor tuple when calling a Level 3 command. The predecessor and successor tuples are already in TS. Therefore, there is no need to provide values for them when storing a tuple. Furthermore, the *read* command does not require the predecessor or successor tuples and the delete command does not return any data for the predecessor or successor tuples. Therefore, it is sufficient to pass the *primary uid* values for the predecessor and successor. Given the *primary uid* values, the TSM can verify that those nodes have not changed since the process attempting the Level 3 operation read them.

We have shown that the Level 3 *insert*, *delete,* and *update* operations only require the *primary uid* values of the predecessor and successor tuples. However, because we must fully specify the target tuple and provide a TSAK, we still need to resolve the syntax for specifying a TSAK, the required *primary uid* values, and all of the fields of the target tuple. One solution would be to list the TSAK, followed by the *primary uid* of the predecessor tuple first, followed by all of the fields of the target tuple, followed by the *primary uid* of the successor tuple all in one comma separated parameter list. This solution is viable, however, it has the undesirable effect of seemingly blending the TSAK and the *primary uid* values of the predecessor and the successor into the fields of the target tuple. The only thing that differentiates TSAK and the *primary uid* values from the fields of the target tuple is their position in the list. The Level 3 *read* operation does not require the *primary uid* of the predecessor and successor tuples, however, the same issue applies with respect to the TSAK and the fields of the target tuple.

Rather than leaving parameter position as the only distinguishing feature between the TSAK, the *primary uid* values and the parameters of the target tuples, we adopt the following syntax for the *insert*, *delete*, and *update* operations. The TSAK and the *primary uid* value of the predecessor tuple are the first two parameters of the Level 3 operation. The *primary uid* value of the successor tuple is the last parameter. The fields of the target tuple are specified in a comma-separated list enclosed in parentheses. The comma-separated list is the third parameter of the Level 3 operation. In the case of the *read* operation, the TSAK is the first parameter. The second parameter is a comma-separated list of the target tuple fields enclosed in parentheses to distinguish them from the TSAK. The partial BNF grammar below (Figure 5.12) clarifies the new syntactical convention required for the Level 3 LDS operations.

```
  <command> →  <rdInst> | <ldsInst>
   <rdInst> →  read ( <ident> , ( <paramList> ) ) ;
  <ldsInst> →  <ldsOp> ( <ident> , <ident> , ( <paramList> ) , <ident> ) ;
    <ldsOp> →  delete | insert | update
 <paramLst> →  <param>, <paramList> | <param>
    <param> →  <formal> | <ident>
   <formal> →  ? <ident>
    <ident> →  Any legal C language identifier
  <literal> →  Any literal value legal in the C language
```

Figure 5.12 – Partial BNF grammar for Level 3 LDS operation syntax.

At Level 1 and Level 2, the successor *uid* was passed to TS as a field of the tuple. At Level 3, the tuple is but one of the parameters of operation. If the successor *uid* is passed as a parameter to the operation, does it also need to be in target tuple? On the other hand, if the successor *uid* is in the target tuple, does it also need to be passed as a parameter to the operation? We adopt the position that the successor *uid* will be passed as a field in the target tuple, as well as a parameter to the Level 3 operation being called. The code segment below shows an example of the results of this decision.

```
rCode = insert(key,predUID,(^targetUID,10,"example",succUID),succUID);
```

At this point, the reader probably wonders why we decided to pass the successor *uid* as both a parameter of the operation and as a field of the target tuple. The reason for this seeming redundancy is not complicated, albeit not immediately apparent. The comma-separated list of parameters comprising the target tuple, even though embedded in syntax that is anything but traditional, follows traditional GC semantics. The semantics of traditional GC do not attach any meaning to the position of parameters within the parameter list. If we were to attempt to extract the successor *uid* from the tuple fields, rather than having it passed as an argument to the command, then we would have to require that the successor *uid* be placed in a particular position within the list of tuple fields. The alternative to that would be to adopt another special symbol, similar to the "^" symbol of the *primary uid*, in order to uniquely identify the successor *uid* from other *uid* values that might appear in the tuple. In either case the requisite changes to traditional GC semantics outweigh the benefit of extracting the successor *uid* from the target tuple, thereby avoiding passing it as a parameter to the operation.

On the flipside of the coin, we could omit the successor *uid* from the target tuple and only specify it as the last parameter of the operation. This option leads to other problems. For example, where is the successor *uid* stored when the node is placed in TS? Is it the first parameter in the tuple, the last parameter in the tuple, or is it stored externally to the tuple? It does not take a great deal of imagination to envision the deleterious effects of this decision on the traditional GC tuple-matching semantics. Therefore, to prevent these issues from becoming problems, the programmer must place the proper successor *uid* in the tuple so that the node is linked into the list properly. In addition, the programmer also passes the successor *uid* to the *insert* operation so that the TSM can verify that the successor node has not been modified.

Given that Level 3 operates in a competitive concurrent environment, the suspicious reader might have a problem with the decision that requires the programmer to specify the successor *uid* as a field of the tuple and also as a parameter of the *insert* operation. The argument against this maintains that the programmer could pass one *uid* in the successor *uid* field of the Level 3 function and a different *uid* in the tuple itself. If this were to be permitted, a rogue process could hijack the LDS by specifying a valid successor *uid* as the parameter to the *insert* operation and a completely different *uid* in the tuple itself. The following example illustrates the nature of the problem.

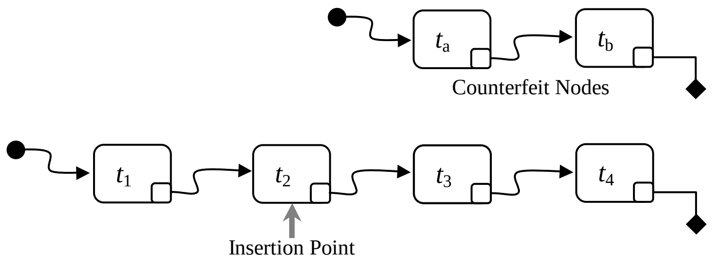


Figure 5.13 – Two LDS in TS with insertion point tuple identified.

The scenario begins with two LDS in TS. Figure 5.13 shows the target LDS with the insertion point tuple identified. The figure also shows another LDS that has been constructed in TS for the purpose of hijacking the first LDS. To insert a new node between tuples t_2 and t_3, the process would execute the following Level 3 *insert* command.

```
insert(tsak t₂, (t_new), t₃);
```

In the benign version of this command, t_{new} contains a pointer to the proper successor tuple – t_3. Figure 5.14 illustrates the way the new node should be linked into the LDS.

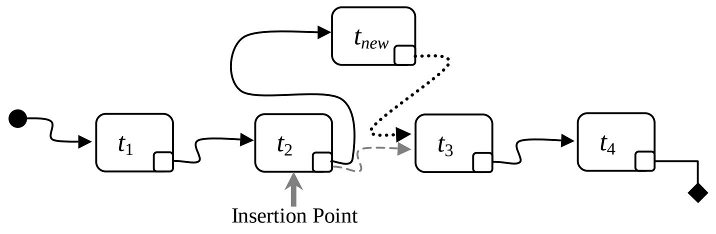


Figure 5.14 – LDS with insertion point tuple identified and new node shown.

The gray, dashed pointer shows the former relationship between tuple t_2 and tuple t_3. The pointer from t_{new} to t_3 is shown as a dotted line to highlight the area where list is vulnerable to possible hijacking attempts.

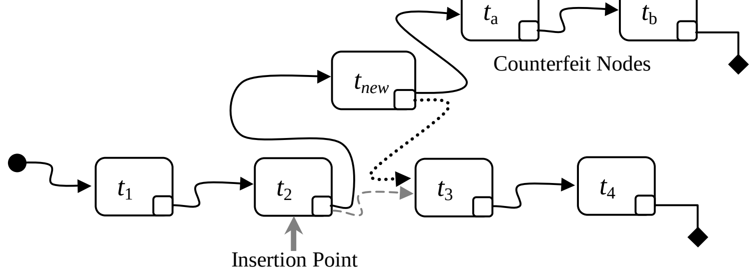


Figure 5.15 – LDS "Hijacked" during insertion by rogue process.

To hijack the list, the malevolent process executes the same command. However, instead of t_{new} pointing to t_3, it points to t_a. Figure 5.15 illustrates the results of this action by the rogue process.

The dotted pointer from tuple t_{new} to tuple t_3 shows how the link should appear. In fact, the view of the LDS that the TSM has, based upon the predecessor *uid* and successor *uid* that was provided to the *insert* operation is consistent with the path from t_2 to t_{new} on to t_3. However, that parameter is only used to verify that the successor node has not changed. The actual link from the new node to its successor node exists within the target tuple itself. Therefore the successor node, after the *insert* operation, is t_a.

Recall that the TSM cannot look into the tuple and be able to distinguish the *uid* of the successor tuple from any other *uid* values that might appear within the tuple. How, then, can we prevent this problem? The key to the problem is that, given the *uid* of the successor tuple as a parameter to the *insert* operation, the TSM can scan the individual fields of the target tuple for a *uid* that matches the *uid* specified as the successor *uid* in the parameter list of the *insert* operation. If the TSM finds a *uid* value within the tuple that matches the *uid* value passed as the last parameter to the *insert* operation, then the TSM can proceed with the operation. Otherwise, by performing this simple check, the TSM can detect the insidious intentions of the rogue process and abort the insertion of the new node. This prevents the problem and alleviates the well-justified concerns of the suspicious reader.

5.8 – What are the semantics of each of the Level 3 LDS operations?

With the syntactical concerns alleviated, we now move on to defining the high-level LDS operations and illustrating their behavior.

5.8.1 – The *insert* operation

To insert a node, the process traverses the list until both the predecessor and successor nodes have been read. The process stores the *primary uid* values of these nodes, in local variables, so that they can be passed to the *insert* operation when it is called. The *insert* operation stands out amongst the other Level 3 operations in that it is the only operation that takes, as its third parameter, a tuple that is not already in TS. Therefore, it requires that the programmer construct the node to be inserted into the list.

The first step in constructing a new node at Level 3 is identical to the first step in constructing a new node at Level 1 and Level 2. The programmer begins by requesting a new *uid* value from the TSM to serve as the *primary uid* of the new node. After the TSM returns the new *uid* value, the next step is to call the *insert* operation. The operation requires the TSAK, the *primary uid* of the predecessor node, the comma-separated parameter list of the target tuple – including the newly requested *primary uid* and finally, the *primary uid* of the successor node. The code segment below shows an example of a call to the *insert* operation. The example assumes that the predecessor and successor nodes have been read from TS and their *primary uid* values are stored in the local variables predUID and succUID. The example further assumes that the TSAK returned by the *connect* operation is stored in the variable key.

```
rCode = insert(key,predUID,(^targetUID,10,"example",succUID),succUID);
```

When the call to the *insert* operation executes, the following sequence of events is carried out on behalf of the process. First, the *reserve* operation is called in order to secure the predecessor node and the successor node for exclusive access by the process. If the *insert* operation successfully reserves the predecessor and successor tuples, then each must be checked to ensure that they have not been modified since the current process read them. The *insert* operation accomplishes this by calling the *modified* operation. If both tuples remain unchanged, then the *insert* operation removes the predecessor tuple from TS. The pointers are updated so the predecessor node points to the target node and the target node points to the successor node. The *insert* operation then stores the target node in TS and it returns the predecessor node to TS. The *insert* operation then calls the *release* operation on the predecessor and successor nodes to remove the exclusive access restrictions from those nodes.

5.8.2 – The *delete* operation

The process of deleting a node using the Level 3 *delete* operation begins by traversing the LDS until the predecessor node, the deletion target, and the successor node are read. After the nodes comprising the *critical range* are read, the process calls the *delete* operation. As the *delete* operation executes, the following actions take place. First, the *delete* operation calls the *reserve* operation to reserve the predecessor and successor tuples for exclusive access. If the *reserve* operation succeeds, the *delete* operation then removes the deletion target from TS without reserving it. Tuples are only reserved if they are either remaining in TS or if they will be returned to TS. The target tuple is, clearly, not going to be returned to TS; therefore it is not reserved for exclusive access prior to its removal from TS.   If the *delete* operation successfully removes the target tuple from TS, then all three tuples comprising the *critical range* are checked by the *modified* operation in order to ensure that they have not been altered since the decision was made to delete the target node. If the nodes have not been altered, then the predecessor tuple is removed from TS and the pointers are updated so that the predecessor points to the successor. The predecessor is then returned to TS and the *release* operation is called to remove the exclusive access rights from the predecessor and successor tuples. The code segment below shows the use of the *delete* operation.

```
rCode = delete(key,predUID,(^targetUID,?intData,?stringData,succUID),
               succUID);
```

5.8.3 – The *update* operation

To update a node, as with deletion, the process begins by reading all three nodes that comprise the *critical range*. After identifying the target node, the programmer saves the *primary uid* values of the predecessor and successor tuples in local variables. The fields of the target node are also bound to local storage so that changes can be made to the data they contain. The programmer makes the desired changes to the target tuple and then calls the *update* operation. The *update* operation begins by attempting to reserve the predecessor tuple, the target tuple, and the successor tuple for exclusive access by the current process. Notice that, unlike the *delete* operation (Section 5.8.2), in the case of the update operation the target tuple must be reserved for exclusive access because it will be returned to TS after the changes have been made. If the *update* operation is successful at reserving all three critical tuples, then the *modified* operation is called to ensure that none of the critical tuples has changed. If the tuples remain unchanged, then the update target is removed from TS. The data of the update target are changed and the target is returned to TS. After returning the target to TS, the reservations on all three tuples are removed via the *release* operation and the update is complete. The code segment below demonstrates the use of the *updateI i*

```
rCode = update(key,predUID,(^targetUID,newIntData,newStringData,succUID),
               succUID);
```

5.8.4 – The *read* operation

At Level 3, list traversal is accomplished with the *read* operation. The *read* operation requires the programmer to specify a template consisting of formal and/or actual parameters. In addition to the template, the *read* operation also requires a valid TSAK so that the TSM can track the sequence of accesses to each node of each LDS in TS. The code segment below shows an example call to the *read* operation.

```
rCode = read(key,(targetUID,?intData,?stringData,?succUID));
```

The syntax of the read operation is similar to that of the *insert*, *delete*, and *update* operations, except that there is no need to pass, as arguments to the *read* operation, the *primary uid* values of the predecessor and successor tuples. The tuple template passed to the *read* operation must obey the same syntax and semantics as those of the other three operations. Therefore, the programmer must include either a formal or actual *uid* parameter in the appropriate position in the template.

The *read* operation returns the data via formal parameters in the template if it finds a match in TS. If there is not a match in TS, the *read* operation behaves like the traditional GC predicate operations – it simply fails and returns with a code that indicates that no match was found. Additionally, if the *read* operation matches a tuple that is reserved by another process, the *read* operation will fail and return a code indicating that the tuple is reserved. The differential failure code allows the programmer to determine if the tuple will be returned to TS or if the tuple has been removed from TS permanently.

Chapter 6 – Prototype Architecture and Implementation Issues

The prototype serves as proof-of-concept for the current research. It also will serve as a research platform for future work related to "GC with Pointers". The purpose is expressly *not* to implement a production-quality GC system. As such, the emphasis is on modularity, flexibility, and extensibility rather than on convenience to the programmer and performance. This chapter begins with an overview of the architecture of the prototype, along with a justification for a few of the more eyebrow-raising design decisions. The remainder of this chapter describes a handful of the most interesting issues encountered during the development of the prototype.

6.1 – Overview of the Prototype Architecture

Figure 6.1 illustrates the major componsents of the prototype, and the dataflow between them. The prototype is a client/server system using the Common Object Request Broker Architecture (CORBA) as the intermediary between the client and the server. We begin the discussion of the prototype architecture on the server-side. The Linda With Pointers (LWP) server consists of two modules implemented in Java – the LWP storage manager and the LWP Server. Together, these two modules serve as the TSM. The LWP storage manager is responsible for storing and retrieving tuples. The LWP Server is responsible for the actual implementations of the methods specified in the CORBA interface. The Object Request Broker (ORB) brokers calls to the server. To ensure cross-platform compatibility, CORBA defines a set of "lowest common denominator" data structures that must be used by applications communicating via the ORB. Data arrive at the server in CORBA-compliant data structures. The LWP server takes this data and constructs instances of tuple parameter objects. Tuples are represented, on the server-side, as vectors of tuple parameter objects.

The vectors of tuple parameters are stored in one of two hash tables (one for *uid* tuples and one for non-*uid* tuples) to facilitate tuple retrieval. For non-*uid* containing tuples, we construct a string representation of the parameter signature of the tuple. Each tuple parameter object contains a character code that indicates the data type of the parameter represented by that object. We build the parameter signature string by concatenating the character codes from the tuple parameter objects. For example, consider a tuple that contains five parameters with the following types: `short`, `long`, `char`, `int`, and `double`. The parameter signature string built from this tuple would be `“SLCID”`. We use the parameter signature string as a key to identify the tuple vectors in the hash table. The vectors of tuple parameters are stored, in another set of vectors, with other tuples that have the same parameter signature. We refer to the vectors containing groups of tuple parameter vectors as buckets. The buckets are stored in the non-*uid* hash table using the parameter signature string as the hash key.

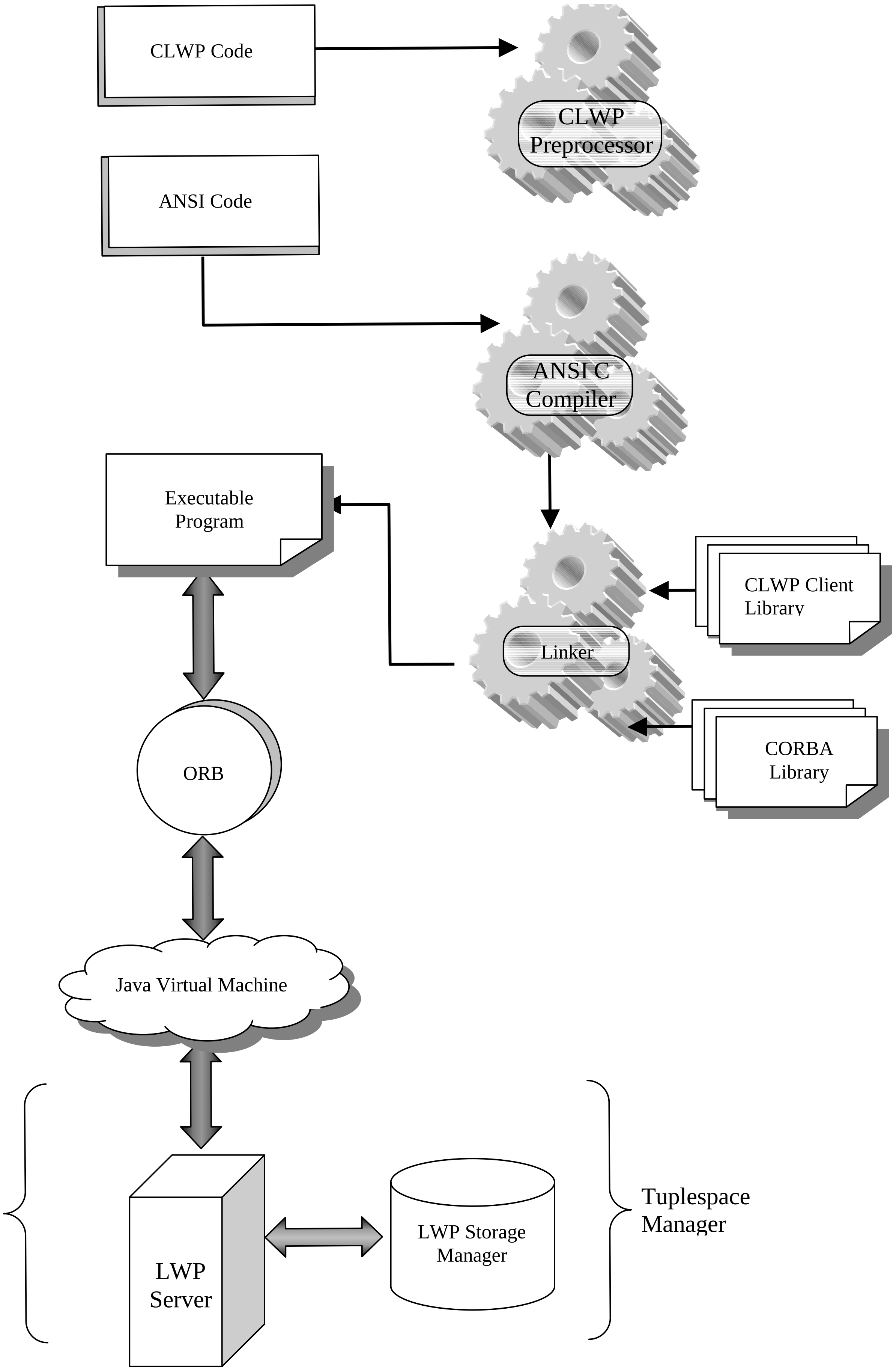


Figure 6.1 – Architectural Overview of Linda With Pointers (LWP).

Retrieval of a non-*uid* tuple involves building a parameter signature string from the types of the parameters in the template. The parameter signature string is used to retrieve the bucket that contains one or more tuples with the same parameter signature. From amongst the tuples stored in the bucket, one tuple is selected non-deterministically. The parameters are passed back from the storage manager to the LWP Server module. The server converts the representations of the data in the Java objects to their corresponding CORBA representations and returns them to the calling routine on the client-side.

The vectors holding the parameters of pointer-containing tuples are stored in a separate hash table. Because no two tuples can have the same *primary uid*, the *primary uid* makes the perfect hash key for this purpose.

Retrieval of *uid* tuples is simpler than retrieval of non-*uid* tuples. First, we use the *uid* as a hash key to retrieve the vector containing the tuple parameters from the hash table. Note that the buckets we used with non-*uid* tuples are not necessary in this case because there can exist, at most, one tuple for each *primary uid*. The LWP server module receives the vector from the storage manager and converts the tuple parameters to an array of CORBA-compliant, equivalent data elements and returns them to the appropriate client-side routine.

The client-side consists of two components. The first is the "C-Linda With Pointers" (CLWP) library. This library contains of a series of C functions, each of which performs one of the GC operations. The library also contains CORBA interface functions that transform the native C data structures into CORBA-compliant data structures and call the appropriate CORBA interface functions.

The second component of the client-side system is a CLWP preprocessor. This preprocessor translates a CLWP program containing TS commands such as *inp*, *rdp*, etc., into an ANSI C program. This is accomplished by replacing the TS commands with calls to the corresponding CLWP library routines.

6.2 – Selected Implementation Issues

While the issues related to the implementation of the prototype are not nearly as numerous as those related to the theoretical underpinnings thereof, a few key issues are interesting and merit discussion.

1. What is the *uid* data type and how is it integrated into the language?

   As with traditional pointers, it is not the specific representation of the value that is important, but rather it is the semantics that are important. For example, traditional pointers are nothing more than integers that refer to a specific, numbered, memory cell. In the case of the *uid* type, we use a 32-bit integer in the prototype for simplicity of implementation. In a production system, the *uid* type could be implemented in such a way so as to ensure a wider range of unique values. What is important about the *uid* type is that the CLWP preprocessor differentiates tuple fields of that type. The differentiation of the *uid* type ensures that the proper CWLP library functions are called, and that the appropriate restrictions on the *uid* type are enforced.

2. Why was Java chosen as the implementation platform for the server-side components?

   Java was chosen as the platform for the server-side components because it has several desirable attributes that fall into line with the stated purpose of the prototype.

   - First, Java is certainly platform-independent. It will run under a variety of operating systems and on a variety of hardware.
   - Secondly, Java includes several convenient data structures such as the java.util.Hashtable and java.util.Vector classes. We would have had to implement and test functionality of these classes ourselves, had we opted to write the server in any of a number of other languages, including C.
   - Finally, the synchronized block construct in the Java programming language offered the ideal mechanism for mutual exclusion within the server code and

storage manager code. Again, this functionality would have to have been implemented and tested, by us, in many other languages.

3. Why is CORBA used?

We opted to use CORBA as middleware between the client-side library, in this case written in C, and the server, which is implemented in Java. The CORBA functionality allowed us to focus on the design of the client and server modules, rather than on developing and testing a communications layer using the raw primitives provided by other languages. Additionally, given that one of the stated purposes of the prototype is to support future research in GC, using CORBA allows clients written in other languages to easily cooperate with the LWP server.

4. Level 3 requires that the TSM be able to indicate whether or not a particular process has read a given tuple since it was last modified. How is this implemented?

The solution is to keep an access journal for each tuple. When a process attempts to modify a tuple, the TSM scans the access list to see if the process in question has read the tuple since the last time it was modified.

5. How can the access journals for each node be prevented from unbounded growth?

The key to preventing unbounded growth of the access journal is the following observation. Once a tuple has been modified, the process performing the modification is the only process that has an accurate picture of the current state of that tuple. Therefore, all previous journal entries are irrelevant and can be purged. Furthermore, it does not matter how many times a process has read a particular tuple since the last time it was modified. The process does not gain any additional information on subsequent reads. Therefore, it is only necessary to keep a record of the most recent read for each process. In this way the size of the access journal is limited to, at most, one record per process per tuple (see Section 5.6.4 for details).

Chapter 7 – Summary and Conclusion

This document has presented what we believe to be a compelling argument for the extension of the GC paradigm to include support for pointers. Having motivated the need for pointers in GC, we systematically outlined and developed a solution to the problem. Beginning with a restricted case of the problem, we extended the C-Linda coordination language to include support for pointers in a non-concurrent (Level 1) environment by introducing the *uid* data type. The success of this solution is demonstrated by constructing a singly linked list. Having solved that problem, we widen the scope of the problem to include cooperative concurrent access to LDS in TS – Level 2. The key to the solution at Level 2 is the cooperative protocol. Of particular interest at Level 2 is the *Reader's Problem,* in which the reading process cannot determine whether a missing tuple has been permanently removed from TS or not. This observation motivated the development of the *lock* operation to disambiguate the removal of *uid* tuples from TS. Finally, at Level 3, we move to competitive concurrency. The obvious solution at this level is to implement operations that follow the cooperative protocol. The problem with the obvious solution is that the locus of the decision point, as to whether or not to abort the operation, cannot reside at the process level as it does at Level 2. The decision point must reside at the system level. This fact leads to the development of the *Critical Range* principle. The critical range principle states that if any of the target, successor, or predecessor tuples has been modified since it was last read, then the operation must fail. The key to the Level 3 solution is the implementation of operations that combine the cooperative protocol of Level 2 with the critical range principle.

7.1 – Work yet to be completed

A significant amount of work has already been completed. However, there are some critical problems yet to be examined.

- Trees – What additional issues must be addressed to include support for trees?

We will identify the issues related to incorporating support for trees into GC. Once the issues have been identified, we will design theoretical solutions, to the problems posed by those issues, by applying what we have learned by solving the problems for linked lists.

- General LDS – Expanding the scope of the problem to general LDS.

  We will apply the lessons we learned, by developing solutions for linked lists and trees, to general LDS. While we will not offer specific solutions that can be applied to any general LDS, we hope to develop a set of guidelines that others can follow to develop solutions to those problems.

7.2 – Timeframe for Completing the Current Research

Work on these problems began during the spring of 2000 and has continued throughout the last three years. Insofar as such a large amount of time and energy have been devoted to developing pointer support for GC, the remaining work – while still significant – will take much less time to complete than that which has already been completed. Therefore, during the spring of 2004, the remaining problems will be solved and significant portions of the written work will be completed. The work will be ready for final defense by late in the fall of 2004.

References

Bal, H. E., & Tanenbaum, A. S. (1988). Distributed programming with shared data. *Proceedings of the IEEE Conference on Computer Languages*, 82-91.

Baratloo, A., Dasgupta, P., & Kedem, Z. M. (1995). Calypso: An environment for reliable distributed parallel processing. *Technical Report: Department of Computer Science, Courant Institute of Mathematical Sciences, New York University.*

Bettini, L., De Nicola, R., Pugliese, R., & Ferrari, G. L. (1998). Interactive mobile agents in X-Klaim. *Proceedings of the 7th IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, 110-117.

Butler, R. M., Leveton, A. L., & Lusk, E. L. (1993). p4-Linda: A portable implementation of Linda. *Proceedings of the Second International Symposium on High-Performance Distributed Computing*. IEEE Computer Society Press.

Campbell D. K. G., Osborne, H. R., & Wood, A. M. (1997). Characterizing the design space for Linda semantics. *Technical Report YCS 277, Department of Computer Science, University of York.*

Carriero, N., & Gelernter, D. (1988). Applications experience with Linda. *Proceedings of the ACM Sumposium on Principles of Programming Languages*, 173-187.

Carriero, N., & Gelernter, D. (1989a). Linda in context. *Communications of the ACM, 32*(4), 444-458.

Carriero, N., & Gelernter, D. (1989b). How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys (CSUR), 21*(3), 323-357.

Carriero, N., Gelernter, D., & Hupfer, S. (1997). Collaborative applications experience with the Bauhaus coordination language. *30th Hawaii International Conference on System Services (HICSS)*, 310-319.

Carriero, N., Gelernter, D., & Leichter, J. (1986). Distributed data structures in Linda. *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, 236-242.

Chang, C., Sussman, A., & Saltz, J. (1995). Support for distributed dynamic data structures in C++. *Technical Report CS-TR-3266*, *University of Maryland Computer Science Department*, *College Park*, *MD*.

Ciancarini, P. (1994). Distributed programming with logic tuple spaces. *New Generation Computing*, *12*(3), 251-284.

De Nicola, R., Ferrari, G. L., & Pugliese, R. (1998). KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, *24*(5), 315-330.

Drucker, R., & Frank, A. (1996). A C++/Linda model for distributed objects. *Proceedings of the 7th Israeli Conference on Computer-Based Systems and Software Engineering (ICCSSE)*, 30-37.

Fleckenstein, C. J., & Hemmendinger, D. (1989). Using a global namespace for parallel execution of UNIX tools. *Communications of the ACM*, *32*(9), 1085-1090.

Gelernter, D. (1985). Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *7(1)*, 80-112.

Gelernter, D. & Bernstein, A. J. (1982). Distributed communication via global buffer. *Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, 10-18.

Gelernter, D. & Carriero, N. (1992). Coordination languages and their significance. *Communications of the ACM*, *55*(2), 97-107.

Hoare, C. A. R. (1975). Recursive data structures. *International Journal of Computer and Information Sciences*, *4*(2), 105-132.

Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, *21*(8), 666-677.

Keleher, P., Cox, A. L., Dwarkadas, S., & Zwaenepoel, W. (1994). TreadMarks: Distributed shared memory on standard workstations and operating systems. *Proceedings of the Winter '94 USENIX Conference*, 115-131.

Kermarrec, Y., Pautet, L. (1994). Ada-Linda: A powerful paradigm for programming distributed Ada applications. *Proceedings of the TRI-Ada Conference*, 438-455.

Lehman, T. J., Cozzi, A., Yuhong, X., Gottschalk, J., Vasudeval, V., Landis, S., et al. (2001). Hitting the distributed computing sweet spot with TSpaces. *Computer Networks, 35*(4), 457-472.

Lehman, T. J., McLaughry, S. W., & Wyckoff, P. (1999). Tspaces: The next wave. *Hawaii International Conference on System Sciences (HICSS-32).* Retrieved April 10, 2003 from IBM Almaden Research Center web site: http://www.almaden.ibm.com/cs/TSpaces/papers/Cluster.ps.Z

Lindholm, T., & Yellin, F. (1996). *The Java Virtual Machine Specification*. Reading: Addison-Wesley.

Murphy, A. L., Picco, G. P., Roman, G. C. (2001). LIME: A middleware for physical and logical mobility. *21st International Conference on Distributed Computing Systems,* 524-533.

Narem, J. E. (1989). An informal operational semantics of C-Linda V2.3.5. *Technical Report YALEU/DCS/TR-839, Yale University Department of Computer Science.*

Nikhil, R. S. (1994). Cid: A parallel, "Shared-Memory" C for distributed-memory machines. *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computing*, 376-390.

Picco, G. P., Murphy, A. L., & Roman, G. C. (1999). LIME: Linda meets mobility. *Proceedings of the 21st International Conference on Software Engineering (ICSE).*

Polze, A. (1993). The object space approach: Decoupled communication with classes. *Research on Communications-Based Systems – Forschungsprojeckte des Graduiertenkollegs Kommunikationsbasierte Systeme, TU Berlin, FB Informatik.*

Seyfarth, B. R., Bickham, J. L., & Arumugham, M. (1993). *Glenda installation and use*. Retrieved April 10, 2003 from the web site of Dr. Ray Seyfarth, Department of Computer Science – University of Southern Mississippi: http://sushi.st.usm.edu/~seyfarth/glenda.tar.Z

Singhal, M., & Shivaratri, N. G. (1994). *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems*. New York: McGraw-Hill, Inc.

Stallings, W. (2003). *Computer Organization and Architecture: Designing for Performance,* (6th ed.). Upper Saddle River: Pearson Education, Inc.

Sun Microsystems (2002). *JavaSpaces Service Specification*, *Version 1.2.1*. Retrieved April 10, 2003 from Sun Microsystems web site: http://wwws.sun.com/software/jini/specs/js1_2_1.pdf

Tanenbaum, A. S. (1999). *Structured Computer Organization*, (4th ed.). Upper Saddle River: Prentice-Hall, Inc.

Wang, H. C., & Feng, M. D. (1999). BaLinda C++: Runtime support for concurrent object-oriented language. *International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN'99)*, 36-41.

Wyckoff, P. (1998). TSpaces. *IBM Systems Journal*, *37(3)*. Retrieved April 10, 2003 from IBM web site: http://www.research.ibm.com/journal/sj/373/wyckoff.html

Yuen, C. K., & Feng, M. D. (1998). BaLinda: A simple parallel programming model. *Massively Parallel Programming Models (MPPM'97)*, 12-22.

Yuen, C. K., & Feng, M. D. (1997). BaLinda: A simple parallel programming model with active objects. *International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN'97)*, 23-29.

Appendix A – Example Linked List Functionality Using Level 1 Operations

At Level 1, there are no restrictions or requirements on the format of tuples with respect to the position of *uid* elements within the tuples. The programmer must build and maintain all pointers to the elements in his or her abstract data structure. The tuple is just an empty container for the elements of the structure. The programmer must fill and organize the data within that container. For discussion purposes, we shall adopt the following conventions for the format of a tuple comprising a node in a traditional singly linked list. Each node in the LDS shall consist of a 3-tuple of the following format:

```
(^uniqueID, data, nextUniqueID);
```

Where *uniqueID* is the *primary uid* of the current node in the linked list; *data* is the data element in the node; and *nextUniqueID* is a reference to the next node in the linked list (i.e., the *primary uid* of the next node in the list).

The following example C code implements functionality to build and maintain a linked list of structures using Level 1 operations. Functions exist to allow the programmer to create a list; insert nodes at the front of the list, the back of the list, in ascending order or in descending order; delete nodes from a list; and change the data value in a node of the list (the ordering of the list, if applicable, is not preserved by this operation).

```
/*****************************************************************************
 *    File: level1.cl                                                        *
 * Version: 1.0                                                              *
 *  Author: Richard L. Neil                                                  *
 *    Date: 7/10/2000                                                        *
 * Revised: 9/13/2001, 9/26/2001                                              *
 ******************************************************************************
 * Copyright Richard L. Neil, 2000,2001.  All Rights Reserved                 *
 ******************************************************************************
 *                                                                            *
 * This file contains functions that will implement typical linked list       *
 * functionality using Level 1 operations. Level 1 operations assume no       *
 * concurrent access to the node structures in tuple space.                   *
 *                                                                            *
 ******************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#define ORDERED_DESCENDING -1
#define ORDERED_ASCENDING   1
#define END_OF_LIST         2
#define FRONT_OF_LIST       3
#define UNORDERED           0
#define EXACT_MATCH         0
#define TRUE                1
#define FALSE               0
```

```
/* Define an example structure to be used as a data container in the linked list */
struct nodeStruct {
   int intData;
} nodeType;

int find_node(char *list_name, nodeType *target_value, int mode, uid *previousNodeUID,
              uid *currentNodeUID, uid *nextNodeUID) {

   int found = FALSE;
   nodeType actual_value;

   rdp(list_name, ?*currentNodeUID);  /* retrieve the uid of the first node by list name */
   if ((*currentNodeUID) && (mode != FRONT_OF_LIST)) {
      *previousNodeUID = NULL;
      do {

         /* retrieve a node with the specified uid, an integer element, and a uid element */
         rdp(*currentNodeUID, ?&actual_value, ?*nextNodeUID);

         /* based upon the specified mode, determine if the desired node has been found */
         if (mode == ORDERED_ASCENDING) {  /* useful for inserting in ascending order */
            if (target_value->intData < actual_value.intData) {
               found = TRUE;
            }
         }else if (mode == ORDERED_DESCENDING) { /* useful for inserting in descending order */
            if (target_value->intData > actual_value.intData) {
               found = TRUE;
            }
         }else if (mode == EXACT_MATCH) {

            /* find a specific node - useful for deletes, updates, etc. */
            if (target_value->intData == actual_value.intData) {
               found = TRUE;
            }
         }
         if (!found) {
            *previousNodeUID = *currentNodeUID;    /* update uid pointers to move down the list */
            *currentNodeUID = *nextNodeUID;
         }
      } while ((!found) && (*currentNodeUID));  /* either we found it or we've run out of list */
   }

   return found;
}
uid create_list(char *list_name) {

   uid list_head = NULL;

   inp(?list_head);              /* retrieve uid for first node */
   if (list_head) {
      out(list_name, list_head); /* store uid of first node under list name */
   }
   return list_head;
}

int delete_node(char *list_name, nodeType *target_value) {

   int found = FALSE;
   uid currentNodeUID = NULL;
   uid nextNodeUID = NULL;
   uid previousNodeUID = NULL;
   uid tempUID;
   nodeType current_value;

   found = find_node(list_name, target_value, EXACT_MATCH, &previousNodeUID, &currentNodeUID,
                     &nextNodeUID);
   if (found) {
      if (previousNodeUID == NULL) {           /* then it must have been the first node */
         inp(list_name, ?tempUID);     /* remove pointer to head of list */
         out(list_name, nextNodeUID);      /* out updated head of list pointer */
         inp(currentNodeUID, ?&current_value, ?tempUID);  /* remove node to be deleted from TS */
         out(^currentNodeUID);                 /* free uid by outting a tuple containing only the uid */
      }else{
         inp(previousNodeUID, ?&current_value, ?tempUID);  /* read in the previous node */

         /* output the previous node with it's next ptr updated */
         out(^previousNodeUID, &current_value, nextNodeUID);
         inp(currentNodeUID, ?&current_value, ?tempUID);  /* remove the node to be deleted from TS */
         out(^currentNodeUID);                 /* free the uid of the node to be deleted. */
      }
   }
   return found;
}


int update_node(char *list_name, nodeType *target_value, nodeType *new_value) {

   int found = FALSE;
   uid currentNodeUID = NULL;
   uid nextNodeUID = NULL;
   uid previousNodeUID = NULL;
   uid tempUID;
   nodeType current_value;

   found = find_node(list_name, target_value, EXACT_MATCH, &previousNodeUID, &currentNodeUID, &nextNodeUID);
   if (found) {
      inp(currentNodeUID, ?&current_value, ?tempUID);    /* remove the node to be changed from TS */
      out(^currentNodeUID, new_value, tempUID);  /* out the updated node to TS */
   }
   return found;
}

int insert_node(char *list_name, nodeType *new_value, int mode) {

   int found = FALSE;
   uid currentNodeUID = NULL;
   uid nextNodeUID = NULL;
   uid previousNodeUID = NULL;
   uid tempUID;
   nodeType temp_value;

   found = find_node(list_name, new_value, mode, &previousNodeUID, &currentNodeUID, &nextNodeUID);
   if (!previousNodeUID && !currentNodeUID && !nextNodeUID) {  /* The list doesn't exist yet */
      currentNodeUID = create_list(list_name);
      if (currentNodeUID) {
         out(^currentNodeUID, new_value, NULL);
         found = TRUE;
      }
   } else if (!previousNodeUID && currentNodeUID) { /* Then we're replacing the head of the list */
      inp(list_name, ?nextNodeUID);  /* remove head of list pointer from TS */
      inp(?tempUID);             /* get a uid for the new node */
      out(^tempUID, new_value, nextNodeUID); /* out new node */
      out(list_name, tempUID);   /* out updated head of list ptr */
      found = TRUE;
   } else if (previousNodeUID && !currentNodeUID && !nextNodeUID) { /* Then we're adding at the end of the
                                                                      list */
      inp(previousNodeUID, ?&temp_value, ?nextNodeUID);    /* remove last node of list from TS */
      inp(?tempUID);                       /* get new uid for new node */
      out(^tempUID, new_value, NULL);           /* out new last node of list */
      out(^previousNodeUID, temp_value, tempUID);      /* out the old end of list with next ptr updated */
      found = TRUE;
   } else {  /* insert between previousNodeUID and currentNodeUID */
      inp(previousNodeUID, ?temp_value, ?nextNodeUID);    /* get the node preceding the new node */
      inp(?tempUID);                       /* get uid for new node */
      out(^tempUID, new_value, currentNodeUID);           /* put the new node in tuple space */
      out(^previousNodeUID, temp_value, tempUID);      /* out the previous node with next ptr pointing to
                                                          new node */
      found = TRUE;
   }
   return found;
}
```

B.9 – Example Linked List Functionality Using Level 2 Operations

The following example C code implements functionality to build and maintain a linked list of structures using Level 2 operations. Functions exist to allow the programmer to create a list; insert nodes at the front of the list, the back of the list, in ascending order or in descending order; delete nodes from a list; and change the data value in a node of the list (the ordering of the list, if applicable, is not preserved by this operation).

```
/******************************************************************************
 *    File: level2.cl                                                         *
 * Version: 1.0                                                               *
 *  Author: Richard L. Neil                                                   *
 *    Date: 1/08/2002                                                         *
 * Revised:                                                                   *
 ******************************************************************************
 * Copyright Richard L. Neil, 2002.  All Rights Reserved.                     *
 ******************************************************************************
 *                                                                            *
 * This file contains functions that will implement typical linked list       *
 * functionality using Level 2 operations. Level 2 operations assume          *
 * cooperative concurrent access to the node structures in tuple space.       *
 *                                                                            *
 ******************************************************************************/

#include <stdlib.h>
#include <stdio.h>

#define ORDERED_DESCENDING -1
#define ORDERED_ASCENDING   1
#define END_OF_LIST         2
#define FRONT_OF_LIST       3
#define UNORDERED           0
#define EXACT_MATCH         0
#define TRUE                1
#define FALSE               0
```

```
/* Define an example structure to be used as a data container in the linked list */
typedef struct nodeStruct {
   int intData;
} nodeType;

int read_tuple(uid *previousNodeUID, uid *currentNodeUID, uid *nextNodeUID,
                   nodeType *data) {

/* This function will attempt to read a tuple from TS.  If the tuple is locked
   the function will continue to attempt to read it until the tuple is either
   returned by the TSM or removed by another process.  If the tuple is removed
   from TS, the function will attempt to back up to the previous node, get the
   next link from that node and return the new successor to the previous node
   to the calling function.  If the attempt to recover fails, the function
   will return FALSE.  Otherwise, the function returns TRUE.

   The function assumes that currentNodeUID is the successor to the previousNodeUID.
   If the specified currentNodeUID cannot be found then the attempt will be to
   find and return the current successor to previousNodeUID.
*/

   int success = FALSE;
   TSReturnCode rcode = LOCKED;
   nodeType tempData;

   /* Attempt to read the tuple identified by currentNodeUID */
   while ((rcode != SUCCESS) && (rcode != REMOVED)) {
      rcode = rdp(*currentNodeUID, ?data, ?*nextNodeUID);
   }
   /* When we get to this point, either the tuple was read or the tuple
      was removed from TS */
   if (rcode == SUCCESS) {
      success = TRUE;
   }else{ /* tuple was removed from TS */
      if (*previousNodeUID) { /* Attempt to back up to the previous node */
         rcode = LOCKED;
         memset(&tempData, 0, sizeof(nodeType));
         /* attempt to read tuple identified by previousNodeUID */
         while ((rcode != SUCCESS) && (rcode != REMOVED)) {
            rcode = rdp(*previousNodeUID, ?&tempData, ?*nextNodeUID);
         }
         /* when we get to this point, either the previous node was returned
            or the previous node was removed */
         if (rcode == SUCCESS) {
            /* Attempt to read the new tuple identified by nextNodeUID */
            rcode = LOCKED;
            *currentNodeUID = *nextNodeUID;
            while ((rcode != SUCCESS) && (rcode != REMOVED)) {
               rcode = rdp(*currentNodeUID, ?data, ?*nextNodeUID);
            }
            /* when we get to this point, either we have the new
               successor to previous node or the new successor was removed. */
            if (rcode == SUCCESS) {
               success = TRUE;
            }  /* else no recovery is possible */
         } /* else no recovery is possible */
      } /* else no recovery is possible */
   }
   /* at this point, if SUCCESS == TRUE then the successor to previous node
      was found one way or another.  If SUCCESS is FALSE, then the successor to
      the previous node could not be found */
   return SUCCESS;
}

int in_tuple(uid *currentNodeUID, nodeType *data, uid *nextNodeUID) {
    TSReturnCode rcode = LOCKED;

    while ((rcode != SUCCESS) && (rcode != REMOVED)) {
       rcode = inp(*currentNodeUID, data, *nextNodeUID);
    }
    return (rcode == SUCCESS);
}

int lock_tuple(uid *currentNodeUID, nodeType *data, uid *nextNodeUID) {
    TSReturnCode rcode = LOCKED;
    while ((rcode != SUCCESS) && (rcode != REMOVED)) {
       rcode = lock(*currentNodeUID, data, *nextNodeUID);
    }
    return (rcode == SUCCESS);
}

int find_node(uid *list_id, char *list_name, nodeType *previous_value,
              nodeType *target_value, nodeType *actual_value, int mode,
              uid *previousNodeUID, uid *currentNodeUID, uid *nextNodeUID) {

   int found = FALSE;
   TSReturnCode rcode = LOCKED;

   while ((rcode != SUCCESS) && (rcode != REMOVED)) {
       /* retrieve the uid of the first node by list id and list name */
       rcode = rdp(list_id, list_name, ?*currentNodeUID);
   }
   /* when we get to this point, either the head of the list has been found or
      the list has been removed from TS */
   if (rcode == SUCCESS) {
      if ((*currentNodeUID) && (mode != FRONT_OF_LIST)) {
         *previousNodeUID = NULL;
         do {

            /* retrieve a node with the specified uid, an integer element, and a uid element */
            rdp(*currentNodeUID, ?actual_value, ?*nextNodeUID);
            if (read_tuple(previousNodeUID, currentNodeUID, nextNodeUID, actual_value)) {
            /* then proceed with list traversal to find desired node */

               /* based upon the specified mode, determine if the desired node has been found */
               if (mode == ORDERED_ASCENDING) {  /* useful for inserting in ascending order */
                  if (target_value->intData < actual_value->intData) {
                     found = TRUE;
                  }
               }else if (mode == ORDERED_DESCENDING) { /* useful for inserting in descending order */
                  if (target_value->intData > actual_value->intData) {
                     found = TRUE;
                  }
```

```
                }else if (mode == EXACT_MATCH) {

                   /* find a specific node - useful for deletes, updates, etc. */
                   if (target_value->intData == actual_value->intData) {
                      found = TRUE;
                   }
                }
                if (!found) {
                   *previousNodeUID = *currentNodeUID;    /* update uid pointers to move down the list */
                   *currentNodeUID = *nextNodeUID;
                   if (previous_value) {
                      /* if a pointer was passed in for previous_value, store
                         the current node data before moving to next node */
                      memcpy(previous_value, actual_value, sizeof(nodeType));
                   }
                }
             }else{
                /* there was an unrecoverable error reading the tuple */
                *currentNodeUID = NULL;  /* the value of currentNodeUID is invalid so don't return it */
             }
          } while ((!found) && (*currentNodeUID));  /* either we found it or we've run out of list */
       }else if (*currentNodeUID) {
          found = TRUE;
       }
    }  /* else the list is removed and we cannot proceed */
    return found;
 }

 int delete_node(uid *list_id, char *list_name, nodeType *target_value) {

    int found = FALSE;
    uid currentNodeUID = NULL;
    uid nextNodeUID = NULL;
    uid previousNodeUID = NULL;
    uid tempUID;
    nodeType current_value;  /* value of target node when found during traversal */
    nodeType actual_value;
    nodeType previous_value;
```

```
TSReturnCode rcode;

memset(&current_value, 0, sizeof(nodeType));
memset(&actual_value, 0, sizeof(nodeType));
memset(&previous_value, 0, sizeof(nodeType));
found = find_node(list_id, list_name, target_value, &previous_value, &current_value,
                  EXACT_MATCH, &previousNodeUID, &currentNodeUID, &nextNodeUID);
if (found) {
   if (previousNodeUID == NULL) {           /* then it must have been the first node */
      rcode = LOCKED;
      while ((rcode != SUCCESS) && (rcode != REMOVED)) {
          rcode = lock(list_id, list_name, ?tempUID);
      }
      if (rcode == SUCCESS) {
          /* remove list header record.  No need to loop on status because
             we have already SUCCESSfully locked the tuple so it must be
             ok to remove it from TS */
          inp(list_id, list_name, ?tempUID);
          /* Check to make sure that the list header still points to the node
             targeted for deletion */
          if (tempUID != currentNodeUID) {
              found = FALSE;
          }else{  /* proceed with deletion */
             /* remove, but do not lock, node to be deleted from TS */
             if(in_tuple(&currentNodeUID, &actual_value, &tempUID)) {
                 /* compare actual_value to current_value to see if changes have
                    occurred since the decision was made to delete this node */
                 if (!memcmp(&actual_value, &current_value, sizeof(nodeType))) {
                     out(^list_id, list_name, nextNodeUID);      /* out updated head of list pointer */
                     out(^currentNodeUID);  /* free uid by outting a tuple containing only the uid */
                 }
             }else{
                 /* an unrecoverable error occurred */
                 found = FALSE;
             }
          }
          (void)unlock(*list_id);
      }else{  /* header record could not be locked */
```

```
                found = FALSE;
            }
        }else{
            /* Lock the deletion point tuple (i.e., the pred of the node to be deleted */
            if (lock_tuple(&previousNodeUID, &actual_value, &nextNodeUID)) {
                /* compare actual_value to previous_value and nextNodeUID to
                   currentNodeUID to see if changes have occurred since the
                   decision was made to delete the successor to this node */
                if ((!memcmp(&actual_value, &previous_value, sizeof(nodeType))) &&
                    (nextNodeUID == currentNodeUID)) {
                   /* then nothing is changed since the decision was made to delete the
                      successor to this node.  It is especially important to check that
                      the nextNodeUID and currentNodeUID are the same.  Otherwise, the
                      node we want to delete has either been deleted or another node
                      has been inserted infront of it */

                   /* Remove from TS, but do not lock, the deletion target */
                   if (in_tuple(&currentNodeUID, &actual_value, &nextNodeUID)) {
                      /* then we must compare the current value of the deletion target
                         to its value at the time the decision to delete was made */
                      if (!memcmp(&actual_value, &current_value, sizeof(nodeType))) {
                         /* proceed with the deletion by removing the pred of the
                            deletion target from TS) */
                         if (in_tuple(&previousNodeUID, &actual_value, &tempUID)) {
                             /* output the previous node with it's next ptr updated */
                             out(^previousNodeUID, &current_value, nextNodeUID);
                             out(^currentNodeUID);                  /* free the uid of the node to be deleted.
*/

                         }else{
                             /* an unrecoverable error occurred */
                             found = FALSE;
                         }
                      }else{
                         /* an unrecoverable error occurred */
                         found = FALSE;
                      }
                   }else{
                      /* an unrecoverable error occurred */
                found = FALSE;
             }
          }else{
             /* an unrecoverable error occurred */
             found = FALSE;
          }
          (void)unlock(previousNodeUID);
       }else{
          /* an unrecoverable error occurred */
          found = FALSE;
       }
    }
  }
  return found;
}

int update_node(uid *list_id, char *list_name, nodeType *target_value, nodeType *new_value) {

  int found = FALSE;
  uid currentNodeUID = NULL;
  uid nextNodeUID = NULL;
  uid previousNodeUID = NULL;
  uid tempUID;
  nodeType current_value;
  nodeType actual_value;

  found = find_node(list_id, list_name, NULL, target_value, &current_value, EXACT_MATCH,
                    &previousNodeUID, &currentNodeUID, &nextNodeUID);
  if (found) {
     /* lock node to be updated */
     if (!lock_tuple(&currentNodeUID, &actual_value, &tempUID)) {
        /* an unrecoverable error occurred */
        found = FALSE;
     }else{
        /* compare current_value to actual_value and nextNodeUID to tempNodeUID
           to determine if changes have been made since the decision was made to
           update this node */
        if ((!memcmp(&current_value, &actual_value, sizeof(nodeType))) ||
                (tempUID != nextNodeUID)) {
               /* an unrecoverable error occurred */
               found = FALSE;
            }else{
               /* remove the node to be changed from TS */
               if (!in_tuple(&currentNodeUID, &current_value, &tempUID)) {
                  /* an unrecoverable error occurred */
                  found = FALSE;
               }else{
                  /* out the updated node to TS */

                  out(^currentNodeUID, new_value, tempUID);
                  found = TRUE;
               }
            }
            unlock(currentNodeUID); /* unlock the tuple so other processes can access it */
        }
      }
      return found;
   }

   int insert_node(uid *list_id, char *list_name, nodeType *new_value, int mode) {

      int found = FALSE;
      uid currentNodeUID = NULL;
      uid nextNodeUID = NULL;
      uid previousNodeUID = NULL;
      uid tempUID;
      nodeType current_value;  /* value of target node when found during traversal */
      nodeType actual_value;
      nodeType previous_value;
      TSReturnCode rcode;

      found = find_node(list_id, list_name, &previous_value, new_value, &current_value,
                        mode, &previousNodeUID, &currentNodeUID, &nextNodeUID);

      if (!previousNodeUID && !currentNodeUID && !nextNodeUID) {  /* The list doesn't exist yet */
      inp(?*list_id);               /* retrieve uid to identify list header record */
      if (*list_id) {
         inp(?currentNodeUID);              /* retrieve uid for first node */
         if (currentNodeUID) {
            /* No need to lock anything here because nothing has been placed
               in TS yet so no other process could possibly be modifying this node */
            out(^currentNodeUID, new_value, NULL);    /* store the fisrt node of the lsit */
            /* store uid of first node under list_id & list name */
            out(^list_id, list_name, currentNodeUID);
            found = TRUE;
         }else{
            found = FALSE; /* an error occurred getting the new UID for the new node */
         }
      }else{
         found = FALSE; /* an error occurred getting the new uid for the list header record */
      }
   } else if (!previousNodeUID && currentNodeUID) { /* Then we're replacing the head of the list */
      rcode = LOCKED;
      while ((rcode != SUCCESS) && (rcode != REMOVED)) {
         rcode = lock(list_id, list_name, ?tempUID);
      }
      /* when we get to this point, either the header was found or
         this list no longer exists */
      if (rcode != SUCCESS) {
         /* the list no longer exists */
         found = FALSE;
      }else{
         /* check to see that the list header still points to the node before
            which we are to insert the new node */
         if (tempUID != currentNodeUID) {
            /* then someone either deleted the node in question or inserted
               a new node before it */
            found = FALSE;
         }else{
            /* remove list header record from TS  -- it must succeed because
               we have already locked it */
            inp(list_id, list_name, ?nextNodeUID);
            inp(?tempUID);             /* get a uid for the new node */
            out(^tempUID, new_value, nextNodeUID); /* out new node */
            out(^list_id, list_name, tempUID);   /* out updated head of list ptr */
            found = TRUE;
         }
      }
      (void)unlock(*list_id);
   } else if (previousNodeUID && !currentNodeUID && !nextNodeUID) {
      /* Then we're adding at the end of the list */

      /* lock the last node of the list so we can update its pointer to refer
         to the new node */
      if (!lock_tuple(&previousNodeUID, &actual_value, &tempUID)) {
         /* an unrecoverable error occurred */
         found = FALSE;
      }else{
         /* compare actual_value to current_value and verify that the
            next pointer of the locked tuple is still NULL */
         if ((memcmp(&actual_value, &current_value, sizeof(nodeType))) ||
              (tempUID != NULL)) {
            /* an unrecoverable error has occurred */
            found = FALSE;
         }else{
            /* remove the last node of the list from TS */
            if (!in_tuple(&previousNodeUID, &actual_value, &tempUID)) {
               /* an unrecoverable error has occurred */
               found = FALSE;
            }else{
               inp(?tempUID);                        /* get new uid for new node */
               out(^tempUID, new_value, NULL);           /* out new last node of list */
               out(^previousNodeUID, new_value, tempUID);      /* out the old end of list with next ptr
updated */
               found = TRUE;
            }
         }
         unlock(previousNodeUID);
      }
   } else {  /* insert between previousNodeUID and currentNodeUID */
      /* lock the previous node so we can update its pointer to refer to the new node */
      if (!lock_tuple(&previousNodeUID, &actual_value, &tempUID)) {
         /* an unrecoverable error occurred */
         found = FALSE;
      }else{
         /* compare actual_value to current_value and tempUID to nextNodeUID to see if anything has
            changed since the decision was made to insert the new node at this point */
         if ((memcmp(&actual_value, &current_value, sizeof(nodeType))) || (tempUID != nextNodeUID)) {
            /* either the data in the insertion point tuple has changed or a different node has
               been inserted after the insertion point tuple */
            found = FALSE;
         }else{
            /* remove insertion point tuple from TS */

            /* get the node preceding the new node */
            in_tuple(&previousNodeUID, &actual_value, &nextNodeUID);
            inp(?tempUID);                       /* get uid for new node */
            out(^tempUID, new_value, currentNodeUID);           /* put the new node in tuple space */

            /* out the previous node with next ptr pointing to the new node */
            out(^previousNodeUID, temp_value, tempUID);
         }
         unlock(previousNodeUID);
      }
      found = TRUE;
   }
   return found;
}
```