Experimental Comparison of d-Rectangle Intersection Algorithms
Applied to HLA Data Distribution

Mikel D. Petty
University of Central Florida, Institute for Simulation and Training, 3280 Progress Drive, Orlando FL 32826-0544

Amar Mukherjee
University of Central Florida, Department of Computer Science, PO Box 162364, Orlando FL 32816-2362

KEYWORDS: High Level Architecture, data distribution, rectangle intersection.

ABSTRACT: The High Level Architecture (HLA) is a standard for constructing distributed simulations. The Data Distribution Management services of HLA reduce the amount of data delivered to an HLA federate by allowing communications connections to be based on federates’ expressed data production and requirements. At the core of determining which connections to make is a geometric problem: finding the dynamic intersection of d-dimensional rectilinear hyperrectangles in d-space. Four different algorithms for solving that problem are described, including a new one developed through application of a data structure from computational geometry. Those algorithms are then compared in an experiment designed to reveal how well they perform in the specific context of the data distribution application. Both intersection performance and connectivity efficiency results are reported.

1. Introduction

1.1 Overview of this paper

An experimental comparison of algorithms for d-rectangle intersection applied to HLA data distribution is reported in this paper. As background for the comparison, concise descriptions of HLA and data distribution are first given in section 1. Section 2 defines the d-rectangle intersection problem, explains its place within HLA data distribution, provides pointers to some related theoretical results, and describes four practical algorithms for data distribution in the context of d-rectangle intersection (including a new algorithm based on a data structure from computational geometry). An experiment to empirically compare the performance of d-rectangle intersection algorithms when applied to data distribution is laid out in section 3. Finally, the results of the experiment are reported in section 4.

1.2 Background

HLA is a standard for constructing distributed simulations. It will facilitate interoperability between a wide range of simulation types and promote reusability of simulation software. HLA consists of a number of interrelated components. The three that define HLA are: (1) the HLA Rules, which define interoperability and what capabilities a simulation must have to achieve it within HLA [DMSO,1996]; (2) the Object Model Template, which is a object-based methodology for specifying simulation and federation object classes, attributes, and interactions [DMSO,1997c]; and (3) the Interface Specification, a precise specification of the functional actions that a simulation may invoke, or be asked to perform, during an HLA exercise [DMSO,1997b]. The HLA Run-Time Infrastructure (RTI) is a software embodiment of the Interface Specification that implements the actions defined therein, including the transport of data between interoperating simulations. A group of simulations interoperating under HLA is called a federation, one of the simulations is a federate, and a simulation run is called a federation execution.

One category of services defined in the Interface Specification, Data Distribution Management, is intended to reduce the amount of simulation data delivered to a federate during a federation execution. To accomplish the reduction, multidimensional coordinate systems, referred to as routing spaces, with axes corresponding to simulation data (simulation object attributes or other data available in the simulation) are defined. Then federates specify what simulation data they are going to produce (publish) or are interested in receiving (subscribe) by creating rectangular regions, called update and subscription regions, within the routing spaces. When an update region and a subscription region “overlap” (i.e., intersect) in a routing space then simulation data should be delivered by the RTI from the publisher to the subscriber. The data distribution services allow update and subscription regions to be created, modified, and deleted dynamically throughout a federation execution. Each time a region is created, modified, or deleted, the set of regions it intersects may change and if so the data delivery connections used by the RTI must be changed to reflect the new region configuration. [DMSO,1997a] describes
the design of HLA data distribution. Detailed information on various aspects of data distribution is available; see any of [Cohen, 1997], [Morse, 1997], [Olzewski, 1996], and [Van Hook, 1996].

2. Data distribution and $d$-rectangle intersection

2.1 Problem definition

In a $d$-dimensional coordinate space, a $d$-dimensional rectilinear hyperrectangle ($d$-rectangle, for short) is a rectangular subset of the coordinate space, with faces parallel to the planes defined by each pair of axes. A $d$-rectangle is the Cartesian product of $d$ closed intervals, one on each axis of the coordinate space, and can be specified by such a set of intervals. Two $d$-rectangles intersect if and only if they share at least one point. There are several formulations of $d$-rectangle intersection problems, but here we are interested in the problem known as dynamic $d$-rectangle intersection searching, defined as follows: process an arbitrary sequence of insertions, deletions, and queries of $d$-rectangles, where a query is to report all current (inserted and not deleted) $d$-rectangles that intersect the query $d$-rectangle.

2.2 Dynamic $d$-rectangle intersection

The dynamic $d$-rectangle intersection searching problem lies at the core of the HLA data distribution routing process. Routing spaces and the update and subscription regions within them are $d$-rectangles. Regions can be created, modified, and deleted dynamically throughout a federation execution. Those events each correspond to one or more operations (queries, insertions, and deletions) in the dynamic $d$-rectangle intersection searching problem. For example, when an update region is created, the subscription regions it intersects must be determined (a $d$-rectangle query) so that appropriate data delivery connections can be established, and the new region must be recorded for future queries (a $d$-rectangle insertion). Similarly, when a subscription region is created the update regions it intersects must be determined and the region recorded. When a region is deleted a $d$-rectangle query is required to determine which data delivery connections should be broken and a $d$-rectangle deletion is needed to remove the region from the current region data structures. Region modifications are logically equivalent to a deletion followed by a creation. Table 1 shows the correspondence between data distribution events and dynamic $d$-rectangle intersection searching problem operations.

2.3 Related theoretical results

Dynamic $d$-rectangle intersection searching and related problems have been studied in computational geometry, usually with a focus on asymptotic worst-case time and space complexity of algorithms to solve the problems. Table 2 provides a set of pointers to some of those results. Space limits preclude more detailed discussions of their applicability, see [Petty, 1997] for more details. In the table, $N$ is the number of $d$-rectangles stored in the data structure and $K$ is the number of intersections found to be reported.

2.4 Data distribution algorithms

In this section four algorithms for data distribution and dynamic $d$-rectangle intersection are presented. The algorithms are:

1. Brute force
2. Gridded
3. $2^d$-tree
4. $d$ interval trees

Each of the algorithms performs the dynamic $d$-rectangle intersection searching process of data distribution, albeit in different ways. In addition to determining $d$-rectangle intersection, the algorithms must dynamically maintain data structures, also specific to the algorithms, in which the $d$-rectangles which are the update and subscription regions are inserted (when created or modified) and deleted (when deleted or modified). The intersection queries for data distribution events operate on those data structures to find intersections with the regions stored in them. The details of this process vary for each algorithm. Note that once a pair of intersecting regions is found a corresponding communications connection must be established. We assume that is done using multicast groups. With the exception of the Gridded algorithm, that process is quite similar for all of the algorithms and is not part of the intersection computation, so we do not discuss it further, except for the Gridded algorithm.

Algorithms which produce only those communications connections implied by the data distribution events are called exact algorithms; those that produce a superset of the implied connections are called approximate algorithms.
<table>
<thead>
<tr>
<th>Data distribution event</th>
<th>Dynamic $d$-rectangle intersection searching operations</th>
</tr>
</thead>
</table>
| 1. Create Update Region | 1. Query: determine subscription regions that intersect the new update region, to establish communications connections.  
  2. Insertion: add the new update region to the data structure that stores update regions. |
| 2. Create Subscription Region | 1. Query: determine update regions that intersect the new subscription region, to establish communications connections.  
  2. Insertion: add the new subscription region to the data structure that stores subscription regions. |
| 3. Associate Update Region | None; no corresponding dynamic $d$-rectangle intersection searching operations. This event completes communications connections. |
| 4. Modify Region | 1. Query: determine regions of opposite type that intersect region before modification, to break communications connections.  
  2. Deletion: remove the region before modification from the data structure that stores regions of its type.  
  3. Query: determine regions of opposite type that intersect region after modification, to establish communications connections.  
  4. Insertion: add the region after modification to the data structure that stores regions of its type. |
| 5. Delete Region | 1. Query: determine regions of opposite type that intersect region, to break communications connections.  
  2. Deletion: remove the region from the data structure that stores regions of its type. |
| 6. Update Attribute Values | None; no corresponding dynamic $d$-rectangle intersection searching operations. This event sends data via communications connections previously established by the other events. |

Table 1 Data distribution events and dynamic $d$-rectangle intersection searching operations.

**Brute force**
The Brute Force method of dynamic $d$-rectangle intersection searching is the simple expedient of testing a query update (subscription) region for intersection with every current subscription (update) region. The $d$-rectangles are kept in two singly linked lists.

**Gridded**
An existing implementation of data distribution solves the dynamic $d$-rectangle intersection searching problem using a method we will refer to as the Gridded algorithm. It is presented in both [Van Hook, 1996] and [DMSO, 1997a]. In the Gridded algorithm, the routing space is subdivided into a predefined regular grid or array of $d$-dimensional cells. Associated with each cell is a stream, which is a logical multicast group. Simple arithmetic on the bounds of a region suffices to identify the cells that the region overlaps. When a region is created, the cells it overlaps are identified and the creating federate joins the multicast groups associated with those cells. For a deletion the federate deleting the region leaves the associated multicast groups. If an update and subscription region intersect they are certain to overlap at least one cell in common, and thus both will be joined to at least one multicast group in common. Data flows from publishing to subscribing federates via the multicast groups they have both joined.

The Gridded method requires only $O(d + K)$ time, where $d$ is the dimension of the routing space and $K$ is the number of cells overlapped, to compute an intersection. However, we observe two difficulties of the Gridded algorithm as a solution to the dynamic $d$-rectangle intersection searching problem. First, it can produce spurious connections, causing data to be delivered from a publishing federate to a subscribing federate even when their update and subscription regions do not overlap. This occurs when the update and subscription regions overlap the same grid cell without intersecting each other. For example, in Figure 1 regions U1 and S1 do not intersect but they both overlap cell $a$, producing a spurious connection.
<table>
<thead>
<tr>
<th>Problem</th>
<th>Method or Data structure</th>
<th>Complexity</th>
<th>Reference</th>
</tr>
</thead>
</table>
| Static rectangle intersection, $d = 2$                                 | Plane sweep                     | Time $O(N \log N + K)$  
Space $O(N \log N)$     | [Six, 1980]              |
| Static rectangle intersection, $d > 2$                                 | Plane sweep                     | Time $O(2^{d-1} N \log^{d-1} N + K)$  
Space $O(2^{d-1} N \log^{d-1} N)$ | [Six, 1982] |
| Rectangle enclosure searching, $d = 2$                                 |                                 | Time $O(N \log^{d} N + K)$  
Space $O(N)$     | [Lee, 1982]              |
| Rectangle enclosure searching, $d = 2$                                 |                                 | Time $O(N \log N \log N + K \log \log N)$  
Space $O(N)$     | [Gupta, 1995]             |
| Rectilinear line segment intersection (Static and dynamic)             | Layered segment tree             | Preprocessing $O(N \log N)$  
Query $O(\log N + K)$  
Space $O(N \log N)$ | [Vaishnavi, 1982b]       |
| Dynamic $d$-rectangle intersection searching, $d = 2$                  | Layered segment tree             | Query $O(\log^{d} N + K)$  
Space $O(N \log N)$ | [Vaishnavi, 1982b]       |
| Static point enclosure searching, $d > 2$                              | Nested segment trees             | Query $O(\log^{d} N + K)$  
Space $O(N \log N)$ | [Vaishnavi, 1982a]       |
| Dynamic point enclosure searching, $d = 2$                             | Nested segment trees             | Query $O(\log^{d} N + K)$  
Space $O(N \log N)$ | [Vaishnavi, 1982a]       |
| Dynamic point enclosure searching, $d > 2$                             | Multidimensional interval tree   | Query $O(\log^{d+1} N + K)$  
Space $O(N \log N)$ | [Mukherjee, 1996]      |
| Static $d$-rectangle intersection searching                             | Transformation to range searching| Query $O(\log^{d+1} N + K)$  
Space $O(N \log N)$ | [Edelsbrunner, 1983a]    |
| Static $d$-rectangle intersection searching                             | $d$-fold rectangle tree          | Preprocessing $O(N \log^{d} N)$  
Query $O(\log^{d+1} N + K)$  
Space $O(N \log^{d+1} N)$ | [Edelsbrunner, 1983a]    |
| Static $d$-rectangle intersection                                       | $d$-fold rectangle tree          | Time $O(N \log^{d+3} N)$  
Space $O(N \log^{d+3} N)$ | [Edelsbrunner, 1983b]    |
| Dynamic $d$-rectangle intersection                                     | $d$-fold rectangle tree          | Query $O(\log^{d+1} N + K)$  
Space $O(N)$     | [Edelsbrunner, 1985]     |
| Batched dynamic $d$-rectangle intersection searching                    | $d$-fold rectangle tree          | Query $O(\log^{d+1} N + K)$  
Space $O(N)$     | [Edelsbrunner, 1985]     |
| Orthogonal object intersection                                          | Nested segment and range trees   | Query $O(\log^{d} N + K)$  
Space $O(N \log N)$ | [Edelsbrunner, 1981]     |

Table 2 Theoretical research results related to dynamic $d$-rectangle intersection searching.
Second, it can produce redundant connections, when the sets of cells overlapped by the update and subscription regions have more than one cell in common. Again in Figure 1, cells $b$ and $c$ are both overlapped by regions $U2$ and $S2$. Such redundant connections could result in multiple deliveries of the same data to a subscribing federate (unless extra processing is expended to avoid that possibility). Note that connections that are both redundant and spurious are possible. Possibly because of the redundant connection problem the initial implementation of the Gridded algorithm permits only point update regions (points are degenerate $d$-rectangles). It is therefore not a fully general solution to either the dynamic $d$-rectangle intersection searching problem or data distribution as defined in [DMSO,1997b].

$2^d$-tree

[Van Hook,1996] proposes an alternative algorithm for data distribution which solves dynamic $d$-rectangle intersection searching using a method we call the $2^d$-tree algorithm. A routing space is subdivided into spatial subspaces, with each space being divided in half along each of the $d$ axes into $2^d$ subspaces; the subdivision continues recursively to depth $D$, the maximum depth of the tree. The routing space is associated with the root of the $2^d$-tree, and each subspace is associated with a child node of the tree in a recursive fashion.

In the $2^d$-tree structure regions ($d$-rectangles) are stored associated with the lowest level $2^d$-tree node that entirely contains the $d$-rectangle in the subspace associated with the node. A query operation begins at the root and proceeds by reporting all $d$-rectangles stored at the current node and continuing recursively to all children whose associated subspaces intersect the query region. Though coding techniques might increase implementation efficiency, this method has a theoretical worst case query time of $O((2^d)^D + K)$, where $d$ is the number of dimensions of the routing space, $D$ is the maximum depth of the tree, and $K$ is the number of $d$-rectangles reported. Also, as with the Gridded method, this method can return spurious connections. If the query is modified to avoid spurious connections by testing each $d$-rectangle stored in the selected subtree for intersection with the query $d$-rectangle prior to reporting it, the worst case query time becomes $O((2^d)^D + N)$.

d interval trees

The $d$ interval trees algorithm is based on the interval tree, a classic data structure from computational geometry. Originally introduced in [McCreight,1981] and [Edelsbrunner,1980], the interval tree stores 1-dimensional data intervals in a data structure that

---

1 [Van Hook,1996] refers to the data structure used in this method as the “multidimensional binary tree”. In the computational geometry literature “multidimensional binary tree” refers to a substantially different data structure; see [Bentley,1975] or [Preparata,1988]. We instead refer to the [Van Hook,1996] data structure as a $2^d$-tree. It is the data structure more commonly known as a quadtree or octtree in 2- or 3-dimensions respectively.
Figure 2  Flow of the d interval trees algorithm.
supports insertion and deletion operations in \(O(\log N)\) time, where \(N\) is the number of stored intervals, and queries in \(O(\log N + K)\) time, where \(K\) is the number of reported intervals. It is a complex data structure that includes a primary binary tree, a secondary search tree threading nodes of the primary tree, and two AVL [Adel'son-Vel'skii,1962] trees for data storage associated with each node of the primary tree. Each primary node in an interval tree corresponds to a portion of the range of possible data intervals. A data interval is stored at the current node, its left subtree, or its right subtree depending on whether the data interval includes, is to the left of, or is to the right of the midpoint of the interval represented by the node. For more information, [Preparata,1988] gives a good explanation of the interval tree.

The algorithm maintains \(d\) interval trees, one for each dimension of the routing space, and stores \(d\)-rectangles by storing the \(d\) intervals that define the \(d\)-rectangle in the \(d\) interval trees. Note that if two \(d\)-rectangles intersect, they must overlap on all \(d\) dimensions. A \(d\)-rectangle intersection query is computed by performing \(d\) 1-dimensional queries on the \(d\) interval trees to find the stored \(d\)-rectangles that overlap the query \(d\)-rectangle on each dimension. The intersection (in set theoretic terms) of those \(d\) query result sets is the answer to the \(d\)-dimensional query. Figure 2 illustrates the process.

Though the \(d\) 1-dimensional queries require total time in the worst case in \(O(d \log N)\), the intersection step increases the overall time complexity of the query to \(O(N)\). The theoretical results surveyed suggested algorithms with better theoretical performance, but we chose to initially implement this non-optimum algorithm because of its conceptual simplicity and potential for practical optimizations. For example, in the implementation the intersection set is maintained incrementally during the \(d\) queries and the overall query is terminated if the intersection set becomes empty. Nevertheless, the theoretical results suggest that better performance may be possible. We shall say more about this later.

3. Comparison experiment

3.1 Experiment intent

The intent of the experiment described herein is to empirically compare the performance of \(d\)-rectangle intersection algorithms when applied to data distribution. An empirical comparison is necessary for two reasons. First, the analytically derived theoretical worst case time complexity for an algorithm is not always indicative of its practical average case time performance. To find out which algorithm is fastest in the average case, it is often most revealing to implement and run them. Second, of interest is the performance of these \(d\)-rectangle intersection algorithms as applied to data distribution. That suggests testing them with data that is representative of data distribution service invocation sequences. Intersection time and memory requirements are compared, as well as connectivity efficiency resulting from the algorithms.

3.2 Experiment design overview

Figure 3 is a schematic overview of the experiment design. The four algorithms to be compared (Brute Force, Gridded, \(2^d\)-tree, and \(d\) interval trees) were implemented within program \texttt{intersect.c}. They were tested using three different event sequences, where an event sequence is a time ordered sequence of region update, modify, and deletion operations that is intended to reveal how well the algorithms performed under specific circumstances. One event sequence, intended to test algorithm correctness, was prepared manually. Two others were produced by the test data generation program \texttt{generate.c}.

When processing an event sequence the intersection algorithms generated as output a sequence of intersection changes which recorded the regions found to newly intersect with an event region (for a create or modify) or to no longer intersect with an event region (for a delete or modify). The intersection changes produced by \texttt{intersect.c} were input to \texttt{connect.c}, which analyzed the sequence of connectivity changes implicit in the intersection changes and calculated how efficiently the publishers and subscribers were connected.

All three programs, \texttt{generate.c}, \texttt{intersect.c}, and \texttt{connect.c} were implemented in ANSI C. The experiment runs were conducted on a SGI O2 Unix workstation.

3.3 Experiment design details

Event sequences

The programs \texttt{intersect.c} and \texttt{connect.c} were tested for correctness using manually prepared test event sequences and expected results. Two other event sequences, called Small and Large, were intended to
**generate.c**
Generate test event sequences

**intersect.c**
1. Find region intersections
2. Report intersection performance

**connect.c**
Report connectivity efficiency

*Figure 3* Data distribution comparison experiment overview.
represent the sequence of data distribution events that might actually occur during federation executions. The test data generation program generate.c essentially simulated the data distribution portions of typical federation executions. The program was given a maximum number of simulation objects of different categories, with the objects in each category having given characteristics relevant to data distribution. See Table 3 which details those characteristics.

Program generate.c looped over its internal simulation objects, probabilistically instantiating new objects, moving existing objects, and removing existing objects (all internally), and generating the appropriate data distribution events for each action. For example, instantiating a new object produced “create update region” and “create subscription region events”, whereas moving an existing object produced two “modify region” events. Program generate.c assumed a 3-dimensional location-based routing space. It maintained a location for each object, which would change as it moved, and was used as the center of the update and subscription regions for that object.

The Small event sequence was intended to simulate a typical small-to-medium sized (in terms of object count) federation execution, while the Large event sequence was generated from more objects. In addition to the object count, a crucial difference between these two event sequences was the different update region size used by generate.c. In the Small event sequence the update region sizes were the distance the object would move in 5 seconds, whereas in the Large event sequence the update regions were points (as used for the STOW RTI-s implementation [DMSO,1997a]).

**Intersection performance measures**

the intersection computation program intersect.c processed the events in the input event sequence, finding intersections and adding and removing $d$-rectangles from their data structures as called for in the events. While doing so it tracked and reported two measures of performance for the intersection. They were: (1) CPU time, the amount of time spent by the algorithms processing the event sequence and (2) Main memory, the maximum amount of storage needed for the algorithms’ data structures during the execution. Program intersect.c used an event batching cycle so as to avoid having any file input/output take place during the timed portions of its execution.

**Connectivity efficiency measures**

The connectivity program connect.c analyzed the resulting intersection changes files and calculated how efficiently the publishers and subscribers are connected. The connectivity efficiency measures calculated fell into three categories: (1) Spurious connections, connections made between a publishing federate and a subscribing federate when no region ($d$-rectangle) intersection existed; (2) Redundant connections, multiple connections made between a given pair of federates; and (3) Multicast group usage, the number of multicast groups needed to make the connections. Within each category several specific measures were collected.

**Simplifying assumptions**

Certain simplifying assumptions were made in implementing the intersection algorithms and the connectivity analysis. For the Gridded algorithm, the location-based routing space was subdivided into $64 \times 64 \times 4$ cells ($x$, $y$, $z$), with the $x$ and $y$ axes considered to be 1 kilometer per cell, giving a reasonable implied terrain database size of $64 \times 64$ kilometers, and the 4 $z$ axis cells thought of as corresponding to ground level, close air, open air, and space. Both generate.c and connect.c assumed that each object had exactly one update region and one subscription region; connect.c also assumed that each object was simulated by exactly one federate and vice versa. Finally, the HLA definition of data distribution allows regions (both update and subscription) that consist of sets of one or more $d$-rectangles; we assume that all regions consist of a exactly one $d$-rectangle. Changing these assumptions provides opportunities for future work.

**4. Results, conclusion, and future work**

**4.1 Results**

Tables 4 and 5 report the intersection performance and connectivity efficiency results obtained for the four algorithms on the Small event sequence. Tables 6 and 7 give the results for the Large test event sequence. Timing values are for the best single run observed for each algorithm.

For the Small event sequence the Gridded algorithm was the fastest. However, its speed came at the price of large numbers of spurious and redundant connections and a heavy usage of multicast groups. The multicast group usage figures for the Gridded algorithm represent the portion of the overall grid of $64 \times 64 \times 4 = 16,384$ cells overlapped by subscription regions in the event sequence. The $2^d$-tree algorithm also produced spurious connections, though not as many. The Brute force and $d$
interval trees algorithms are exact, producing precisely those connections implied by the event sequence, so they have no spurious or redundant connections. However, on the moderately-sized Small test event sequence the $d$ interval trees was only slightly faster than the Brute force algorithm.

In the Large event sequence the Gridded algorithm remains clearly the fastest. However the logarithmic power of the interval tree becomes apparent in the relative time increases of the $d$ interval trees algorithm versus the Brute force algorithm; the $d$ interval trees algorithm is now much faster than the Brute force algorithm. Examination of the program with a execution profiler showed that the heuristic optimizations of the $O(N)$ intersection portion of the $d$ interval trees algorithm worked for this test event sequence.

As for connectivity efficiency, the use of point update regions in the Large event sequence has entirely eliminated the redundant connections made by the Gridded algorithm, although the number of spurious connections has grown. However, as pointed out earlier, restricting update regions to points is not a general solution to data distribution. The precise values of the connectivity efficiency results are dependent on many of the parameters of the experiment, including region size, cell size, and number of objects, all of which are likely to be different from one federation execution to the next. Nevertheless, the connectivity efficiency results do indicate trends. The results show clearly that the Gridded algorithm pays for its speed by creating numbers of spurious and redundant connections, a problem the exact algorithms do not have. Additional work is needed to quantify the costs on each side of that tradeoff.

### 4.2 Conclusions

HLA has been designated as the standard architecture for defense simulations and thereby takes on considerable significance. Effective and efficient data distribution is crucial to the scalability of HLA; without effective data distribution, simulations will be overwhelmed by the expected volume of incoming data; without efficient data distribution, data delivery will be inordinately delayed by data distribution overhead processing. Thus data distribution is a very real and important practical application.

Of the algorithms compared, the Gridded algorithm is fastest, though in our experiment that algorithm created large numbers of spurious and redundant connections and used large numbers of multicast groups. The $d$ interval trees algorithm had the second best execution time while retaining exact connectivity efficiency. Further execution time improvements in dynamic $d$-rectangle intersection searching are possible through application of available theory.

### 4.3 Future work

The task raises at least as many questions as it answers, a happy circumstance for a researcher. What is the effect of object parameters, such as region size, movement speed, and update frequency, on the connectivity efficiency measures? What effect do algorithm parameters, such as size and number of grid cells (for the Gridded algorithm) or number of tree levels (for the $d$ interval trees algorithm) have on execution speed? How does the one object-one federate assumption, which plainly omits CGF systems, affect the connectivity efficiency results? Many of these parameters can be refined to more closely fit expected federation executions.

Most importantly, what is the tradeoff between more processing load for the individual federates, as necessitated by the filtering required to discard unwanted data delivered on spurious and redundant connections when an approximate algorithm is in use, versus more work for the RTI, as implied by the centralized data structures and greater execution time of the exact algorithms? Is it possible to devise an exact algorithm that does not require centralized processing? All of these questions merit further investigation.
<table>
<thead>
<tr>
<th>Event sequence</th>
<th>Large</th>
<th>Small</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object category</td>
<td>Ground</td>
<td>Close air</td>
</tr>
<tr>
<td>Minimum number of objects in this category simultaneously instantiated</td>
<td>2689</td>
<td>100</td>
</tr>
<tr>
<td>Maximum number of objects in this category simultaneously instantiated</td>
<td>4748</td>
<td>200</td>
</tr>
<tr>
<td>Probability of instantiation, if not instantiated</td>
<td>.035</td>
<td>.035</td>
</tr>
<tr>
<td>Probability of removal, if instantiated</td>
<td>.025</td>
<td>.025</td>
</tr>
<tr>
<td>Movement speed, $xy$ (meters per second)</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>Movement speed, $z$ (meters per second)</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Update region size, $xy$ (meters)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Update region size, $z$ (meters)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Subscription region size, $xy$ (meters)</td>
<td>5000</td>
<td>10000</td>
</tr>
<tr>
<td>Subscription region size, $z$ (meters)</td>
<td>2500</td>
<td>4500</td>
</tr>
</tbody>
</table>

*Table 3 Object category parameters for generate.c.*
### Intersection performance measures

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Brute Force</th>
<th>Gridded</th>
<th>(^2)-tree</th>
<th>(d) Interval Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU time (seconds)</td>
<td>3.850</td>
<td>0.852</td>
<td>56.948</td>
<td>3.434</td>
</tr>
<tr>
<td>Main memory (K bytes)</td>
<td>904</td>
<td>2,536</td>
<td>58,720</td>
<td>972</td>
</tr>
</tbody>
</table>

*Table 4 Intersection performance results for the Small event sequence.*

### Connectivity efficiency measures

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Brute Force</th>
<th>Gridded</th>
<th>(^2)-tree</th>
<th>(d) Interval Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Spurious connections</strong></td>
<td>0</td>
<td>4,497</td>
<td>3,162</td>
<td>0</td>
</tr>
<tr>
<td>Average spurious connections per event</td>
<td>0</td>
<td>1,194</td>
<td>.840</td>
<td>0</td>
</tr>
<tr>
<td>Average spurious connection duration</td>
<td>0</td>
<td>1,821,888</td>
<td>217,519</td>
<td>0</td>
</tr>
<tr>
<td><strong>Redundant connections</strong></td>
<td>0</td>
<td>15,803</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Average redundant connections per event</td>
<td>0</td>
<td>4,196</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Average redundant connection duration</td>
<td>0</td>
<td>1,868,407</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Multicast group usage</strong></td>
<td>281,736</td>
<td>7,841,390</td>
<td>292,068</td>
<td>281,736</td>
</tr>
<tr>
<td>Average multicast groups in use</td>
<td>311</td>
<td>13,568</td>
<td>335</td>
<td>311</td>
</tr>
<tr>
<td>Maximum multicast groups in use</td>
<td>427,018</td>
<td>4,849</td>
<td>218,188</td>
<td>157,154</td>
</tr>
</tbody>
</table>

*Table 5 Connectivity efficiency results for the Small event sequence.*

### Intersection performance measures

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Brute Force</th>
<th>Gridded</th>
<th>(^2)-tree</th>
<th>(d) Interval Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU time (seconds)</td>
<td>427,018</td>
<td>4,849</td>
<td>218,188</td>
<td>157,154</td>
</tr>
<tr>
<td>Main memory (K bytes)</td>
<td>4,932</td>
<td>5,412</td>
<td>88,080</td>
<td>4,304</td>
</tr>
</tbody>
</table>

*Table 6 Intersection performance results for the Large event sequence.*

### Connectivity efficiency measures

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Brute Force</th>
<th>Gridded</th>
<th>(^2)-tree</th>
<th>(d) Interval Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Spurious connections</strong></td>
<td>0</td>
<td>387,473</td>
<td>217,926</td>
<td>0</td>
</tr>
<tr>
<td>Average spurious connections per event</td>
<td>0</td>
<td>8,817</td>
<td>4,959</td>
<td>0</td>
</tr>
<tr>
<td>Average spurious connection duration</td>
<td>0</td>
<td>22,994,773</td>
<td>320,899</td>
<td>0</td>
</tr>
<tr>
<td><strong>Redundant connections</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Average redundant connections per event</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Average redundant connection duration</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Multicast group usage</strong></td>
<td>2,385,746</td>
<td>10,661,346</td>
<td>2,514,245</td>
<td>2,385,746</td>
</tr>
<tr>
<td>Average multicast groups in use</td>
<td>3,087</td>
<td>12,310</td>
<td>3,146</td>
<td>3,087</td>
</tr>
<tr>
<td>Maximum multicast groups in use</td>
<td>2,385,746</td>
<td>10,661,346</td>
<td>2,514,245</td>
<td>2,385,746</td>
</tr>
</tbody>
</table>

*Table 7 Connectivity efficiency results for the Large event sequence.*
5. References


6. Acknowledgments

The authors thank Robert W. Franceschini for invaluable assistance during the optimization process. Work on this project was partially supported by the UCF Institute for Simulation and Training under the Developmental Leave program (Petty) and the UCF Office of the Vice President for Research and Graduate Studies under the College and Center/Institute Research Initiative program (Mukherjee). That support is gratefully acknowledged. We also thank the DSS referees for their useful comments on an earlier version of this paper.

7. Authors’ Biographies

Mikel D. Petty is a Senior Research Computer Scientist at the Institute for Simulation and Training. He has led IST research in High Level Architecture, Emergency Management, and Computer Generated Forces. Dr. Petty received a Ph.D. from the University of Central Florida in 1997, an M.S. from UCF in 1988, and a B.S. from the California State University Sacramento in 1980, all in Computer Science. His research interests are simulation and computational geometry.

Amar Mukherjee is a Professor of Computer Science at the University of Central Florida. Professor Mukherjee received the D.Phil.(Sc.) degree from the University of Calcutta in 1962 at the Institute of Radiophysics and Electronics. Since then his research has been focused on VLSI design, algorithms, and architectures. He has held faculty positions at the University of Iowa, Montana State University, and Princeton University. Professor Mukherjee is a Fellow of the IEEE and served two terms as Editor of the journal IEEE Transactions on Computers.