An Adaptive Interest Management Scheme for Distributed Virtual Environments

Rob Minson and Georgios Theodoropoulos

School of Computer Science,

University of Birmingham, UK

E-Mail: {rzm|gkt}@cs.bham.ac.uk

Abstract—Traditionally Interest Management (IM) in distributed environments has been performed through a ‘top-down’ expression of interest patterns in the model. For many models the interest patterns are complicated or highly dynamic and are not suitable for expression in this way. Top-down expression also forces the model to contain data about the network communication which should strictly be encapsulated by the infrastructure. We propose a system, based on a Distributed Shared Variable abstraction, capable of automatically deriving interest expressions at the infrastructure level in a ‘bottom-up’ manner. We demonstrate through a simulated testbed that for certain access patterns this dramatically improves the precision of the IM system without sacrificing transparency from a modeller perspective.

I. INTRODUCTION

A Distributed Virtual Environment (DVE) can be characterised as a system of simulation nodes such that each node in the system is responsible for the modelling of a subset of a shared environment. An update occurring at a given node is therefore likely to have immediate significance for only a subset of the other nodes in the system. Interest Management (IM) is the process by which one exploits this *interest* of each node to minimise the number of update messages that must be propagated.

IM systems are found in all forms of distributed virtual environment including large-scale distributed simulations [1], [2]; virtual reality systems [3], [4], [5]; and multiplayer networked games [6], [7]. All such systems currently adopt an approach in which nodes can express interest in (‘subscribe to’) some subset of the model, all updates pertaining to this subset are forwarded to the node. The task of formulating these expressions is left to the model and hence we can term this class of IM system *top-down*.

In many situations top-down IM schemes actually provide inefficient solutions to the problem of bandwidth reduction whilst introducing development overhead by making these low-level issues of network communication part of the model specification. We propose a bottom-up framework that uses infrastructure-level observation of the access patterns of the model to automatically and continuously derive and maintain a node’s data subscriptions.

The remainder of this paper is organised as follows. Section II gives a brief survey of the current state of the art in top-down IM schemes. Section III gives a more formal expression of the problem of interest management. Sections IV and V outline

our proposed use of a Distributed Shared Variables abstraction to address this problem through pattern prediction. In sections VI and VII we describe the design and testing of a simplistic example of such a pattern prediction algorithm and section VIII describes experimental results for these tests. Finally sections IX and X discuss related research in this area and the probable direction of this research in the immediate future.

II. TOP-DOWN IM SCHEMES

The following general IM schemes all provide some mechanism by which the model expresses interest patterns for each node. The precise details of these mechanisms effect the precision, transparency and flexibility of the scheme in general.

A. Grid-Based Schemes

These schemes divide the universe of data in a model into an n-dimensional grid of interest cells, requiring individual nodes to subscribe to some set of cells. Updates are sent only between nodes who share some subset of cell subscriptions. The most historically influential grid-based scheme was NPSNET [2], implemented for the Distributed Interactive Simulation protocol (DIS) [8].

The advantages of a grid-based IM scheme are relatively clear. The world is decomposed into fairly intuitive interest landscape where the semantic of an area of interest at model level (i.e. a physical area of space) is easily mapped on to the interest expression that the executive requires. Also, because the decomposition of the world is static, a very close mapping on to multicast groups can be statically defined, creating a highly optimised network layer.

Cell-granularity is the primary issue for grid-based schemes. Too large a cell size creates imprecise interest expressions while too small a size means that fast moving entities provoke thrashing in the network layer by constantly changing their multicast group memberships. These disadvantages of static, grid-based schemes have been explored by the NPSNET group in [9].

B. Class-Based Schemes

The DIS protocol itself [8] performs data exchange between nodes through the propagation of Protocol Data Units (PDUs). These PDUs are split in to classes depending on the information to which they pertain. Nodes in such a scheme

may express their interest through subscription to some set of classes.

This approach is potentially extremely flexible but requires a large amount of machinery at the model level for taking advantage of certain interest patterns. The work of Sorroche and Szulinski [10] demonstrates the complexity involved in enabling large DIS models to take advantage of limited spatial interest regions for bandwidth reduction. This work demonstrates that a purely class-based IM scheme cannot easily express interest in a discrete area of continuous space.

C. Region-Based Schemes

Region-based schemes, such as HLA's Data Distribution Management (DDM) [11], are similar to grid-based schemes in that they require nodes to explicitly subscribe to notional areas of the data-space. Region schemes are more general however, in the sense that they do not assume a static decomposition of the data space but instead allow the nodes themselves to define arbitrary regions of data in which they are interested.

Because both update and subscription regions can be created and modified arbitrarily at runtime and because of the arbitrary dimensionality of regions, DDM is very expressive in comparison to both grid- and class-based schemes. This expressivity does come with a significant overhead, however. Because update-regions may not map directly to subscription-regions, and because of the arbitrary dimensionality of these regions, algorithms for calculating update-groups in region-based IM are costly. Research by Morse on minimising these recalculations [12] and on finding optimal multicast mappings [1] clarifies the problems and offers solutions. However, such solutions generally trade flexibility and transparency for performance.

D. Bespoke Schemes

Outside of the distributed simulation domain most distributed environments use bespoke IM systems tightly coupled to the specific modelling task.

Virtual Reality research has explored very intricate algorithms for visibility based IM [5], [4]. Architectures have also been proposed in [3] and [13] to combine these algorithms with graceful degradation of fidelity in the event of network overload. Such sophisticated techniques generally rely on a large number of model-specific heuristics.

This is also the case with network game engines such as Unreal [6] which relies on an intimate knowledge of the behaviour of entities in the virtual environment for its calculation of a client's 'relevance set'.

In general these 'hard-tuned' systems are clearly inappropriate for flexible distributed simulation platforms where a-priori knowledge of interest patterns is unavailable. More significantly, because these systems rely on these static definitions they are unable to adapt to situations in which heuristics prove inaccurate.

III. LEVELS OF PRECISION

The problem of interest expressions in general is to find some way of expressing a set of data items which will be

used (read) by the modelling process at a node within some time interval. We can define the *precise* sphere of interest $SOI(n, [T])$ to be the set of variables that are accessed (read) by the node n during the notional time interval $[T]$ ¹.

In the schemes reviewed in section II the definition of an interest expression in these terms is in fact a two-stage process. First the model expresses some range of *values* to the IM infrastructure (e.g. a set of object classes, or a range of integer values in a coordinate space). At runtime the infrastructure then evaluates each update message in the rest of the system against this expression and transmits only the matching ones. We can define $u(ie, [T])$ for a given interest expression ie as the set of variables for which at least one update is returned by this procedure over the time interval $[T]$.

If we represent a theoretical interest expression that will evaluate to *true* for *any* update as ∞ then a given interest expression ie in any of the schemes described above is *valid* for a given node n iff

$$u(ie, [T]) \supseteq (u(\infty, [T]) \cap SOI(n, [T]))$$

In other words the expression is valid only if it generates a set of variables which contains all the variables which experienced updates *and* were read by n .

The process of formulating an interest expression is in fact the process of conservatively predicting the size of this intersection. The overlap:

$$u(ie, [T]) - (u(\infty, [T]) \cap SOI(n, [T]))$$

for any such valid expression is the set of variables whose updates have been sent erroneously as a result of too conservative an estimation of the size of the intersection. In the schemes discussed in section II, the size of this set is generally relative to the dynamism of the model and expressivity of the IM scheme.

An immediate solution to eliminate this overlap is to switch from *push*-processing, where updates to a variable at the owner node are immediately sent to other nodes, to *pull*-processing [14], where accesses to variable values cause a request-reply exchange between the 'remote' node and the 'owner' node, while updates at the owner node simply modify the local copy.

The problem with pull-processing is that it trades one overlap for another. A pull performed on some variable v at time t_r is only necessary if a write on v has occurred since the previous read. The overlap in pull-processing is not between one set of variables and another, but between the set of necessary pulls and the set of actual pulls.

An optimal interest management scheme is one which can ensure an empty overlap between the necessary messages and the actual messages. In the case of push-processing this implies that over any time interval:

$$u(ie, [T]) \equiv SOI(n, [T])$$

In the case of pull-processing it implies some scheme whereby queries are only sent if the owner's value has changed since

¹At this abstract level, whether our time system is virtual or real is immaterial.

the last read. Both of these optimal schemes are precluded due to the properties of uncertainty in distributed systems. The remainder of this paper therefore discusses a bottom-up framework capable of automatically deriving a sub-optimal approximation of these schemes.

IV. A PROBLEM STATEMENT BASED ON DSVs

Our framework is based on a runtime analysis of read and write patterns. In order to gain access to these data we make use of Distributed Shared Variables (DSVs) to represent the shared state in a distributed environment. The DSV abstraction is used largely as means of providing the infrastructure with the access information it requires from the model. Note at this point that the framework itself is reliant on no particular distributed consistency model such as those discussed by Mehl and Hammes [15]. The possibilities of extending the scheme for different consistency models are discussed in section X. For the sake of simplicity in this discussion our model is based on the central server approach, where one node in the system is the *owner* of the variable and holds the *master* copy, all other nodes are *replicators* of the variable and hold *proxies*.

We start from the position that both push-processing and pull-processing alone will produce large amounts of redundant messages. For a given model, one algorithm may give less redundancy than another, but even during a single run this balance will change over time, this general pattern scales down to the level of a single DSV during a discrete time interval.

This general idea is illustrated in fig. 1. This figure shows a single master/proxy pair, in the first case the communication is managed by push-processing, in the second by pull-processing. An example sequence of operations at each variable is also presented².

Fig. 1. Messages exchanged between a replicator r and an owner o given a sequence of reads (re) and writes (wr) for both push-processing (above) and pull-processing (below)

²Note that in the central server approach, writes to proxies must *always* be communicated to the owner, while reads on masters are *never* communicated to the replicators. Because of this imperative these operations are a static cost and hence are unimportant for our analysis.

By counting the number messages each processing scheme produces for a given sequence of operations we can determine which would be preferable for the time interval enclosing these operations. From table I it is clear that push processing is preferable over the entire interval $[t_0, t_8]$, but the opposite is true over the interval $[t_0, t_7]$.

Scheme	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	Total
Push	0	0	1	1	1	0	0	0	3
Pull	0	0	0	2	0	0	0	2	4

TABLE I

PUSH VS. PULL PERFORMANCE FOR A GIVEN OPERATION SEQUENCE

This problem of segmenting known sequences to minimise messages is in itself a non-trivial problem as the search space of possible segmentations increases polynomially with the size of the sequence. The problem we face however, is to do so *on the fly* by predicting the evolution of the sequence.

It is at this level that we propose to perform our analysis and it is by modulating a single DSV between push- and pull-processing based on pattern prediction that we propose to automatically effect interest management.

V. THE PUSH-PULL FRAMEWORK

Initially we define the basic machinery of the infrastructure:

- The system consists of a set of *nodes*, all viewing a common set of variables in the form of DSVs. For each DSV, one node is the *owner* and all others are *replicators*.
- A DSV is a single *master* variable at the owner and n *proxies*, one at each replicator.
- Each $\langle \text{master}, \text{proxy} \rangle$ pair represents an update *edge*, this being an abstraction of some virtual channel in a communication system
- At any point in real time an edge is in one of two *update states*, *push* or *pull*.
- The update state is implemented by boolean flags at the master and proxy forming this edge, describing whether writes at the master should be pushed and whether reads at the proxy should be pulled. The update state can be *flipped* from one state to another by a message-based handshake operation.

The task is to define some mechanism for managing the update states to minimise the number of messages. In considering algorithms for this management, understanding the memory access model used by the application is vital. We consider here two of the most common models.

A. Support for a Reference-Space Memory Access Model

A reference-space access model is one in which data items are accessed by specifying some unique identifier for that item, such as the name of a variable. There are a number of different levels at which support can be provided for managing update state given such an access model.

At the level of a *single edge* the task is to define a prediction model for a single operation sequence such as the one given in

section IV, and apply update state flips based on this model, taking into account the cost of the handshake messages. The *demand-push* algorithm outlined in section VI is a simplistic example of such a prediction model.

Moving one level up through the machinery, we can perform a prediction analysis based on correlative behaviour between *groups of edges*. Whilst correlation analyses are relatively expensive, identifying sets of edges whose operation patterns are related may make further prediction cheaper and more meaningful with regard to certain models.

Beyond the level of groups of edges, the prediction of patterns of behaviour begins to rely on some higher-level understanding of the model itself. A prediction model may be constructed at this level, but it is likely that it would have some kind of heuristic component.

This framework in general does have the possible disadvantage of high memory demands. Let v be the number of master variables, and N be the number of nodes in the system, then each node will have to record $v(N - 1)$ update-states (one for each proxy). Depending on the prediction model, prediction data may also have to be recorded in similar quantities. It is clear from this that in very large networks with a large number of local variables at each node, available memory may become a factor.

In such cases it may be necessary to investigate clustering nodes together in push-pull groups, which would decrease the number of update-states for local variables to $(n/s) + s - 2$ for an n node network with a cluster-size of s . However, it is probable that this would incur a cost of increased network messages as the accuracy of the node-to-node interest approximation is worsened. Note that this scheme is comparable to the hierarchical routing structures for shared-state data proposed in [16].

B. Support for an Associative Memory Access Model

If one considers the realities of modelling it is unlikely that data-gathering will take place solely on the basis of individual read operations. Just as important a facility for a model, is the ability to determine the identities of variables whose value matches some predicate (e.g. *the position of all people within 5 yards of point (x,y)*).

This abstract memory access model is the *associative* model and it is present in the push-pull framework in the form of the *range query* operation, it can be expressed more formally by:

$$rq(c, ?id, V \mid or \mid [x, y])$$

Where c is a given variable class, V is a non-empty set of values or predicates and $[x, y]$ is a value interval in the domain c such that $x \leq y$. The $?id$ symbol denotes that these commands will return a set of id values of variables whose value is equal to the given value.

Upon receiving a range query invocation we can simply perform a read of each variable of class c , comparing the returned value to the predicate and returning the satisfying set. This algorithm - the serial-read algorithm - sends a number

of messages equal to twice the number of proxies of class c whose update state is currently set to *pull*.

The operation can be optimised by defining an extra virtual edge between each node which we term the *bulk-read* edge. This edge is specifically for performing range-queries. When a range query is executed by the model, the following algorithm is executed by the infrastructure:

- 1) The querying node r passes the rq parameters down the bulk-read edges to all other nodes O such that there exists an edge (o_i, r) with a variable-class of c , and an update-state of *pull*.
- 2) Upon receiving the query o_i checks each variable of class c whose update-edge to r is set to *pull*, for equality with the query-predicate and returns the satisfying set S of $\langle var_id, value \rangle$ tuples.
- 3) Meanwhile r performs the same operation locally, by evaluating a satisfying set from its own master variables of class c and from all proxy variables of class c where the update state is set to *push*.
- 4) Once all satisfying sets have been received from O at r they are combined and returned as the result of the query.

Using the bulk-read algorithm one reduces the number of messages for each range query in comparison to the serial-read algorithm. A range query at node n_i will send one message for each node n_j (where $j \neq i$) such that there exists a proxy at n_i with an update state of *pull* and a class of c . Because a serial-read would send a message for each of these proxies anyway, the worst performance of the bulk-read algorithm is exactly equal to the serial-read algorithm. Bulk-reads also have a scaling property since the evaluation of predicate equality is evenly distributed across all nodes and executed in parallel.

The bulk-read algorithm provides reasonable performance support for associative memory access. Section X discusses the possibility of using more advanced pattern prediction techniques to optimise associative access patterns in the same way as reference-space access patterns.

VI. THE DEMAND-PUSH ALGORITHM

This section presents a basic algorithm of the type proposed in section V. We assume a reference-space memory access model only, so range queries can never be issued by the model. The algorithm works by managing the update-state of a single edge over time.

The demand-push algorithm works in a similar way to demand-paging in virtual memory systems. When a read is performed upon a proxy, a timer T is set running on the proxy, taking a period of length k to reach 0. While $T > 0$ the proxy's edge is left in push mode, when T reaches 0 the replicator node sends a message flipping the edge back to pull-mode. If a read occurs on a proxy in push-mode this is not communicated to the owner, T is simply reset to k .

The demand-push algorithm is likened to a demand-paging algorithm in the sense that remote reads are taken to be analogous to page-faults in a paged-memory system. In both

cases the caching of recently accessed localities is an attempt to minimise the number of future faults.

In both cases caching also incurs a cost, in the case of demand-paging the cache is of a limited size, so for every page cached another must be selected for removal. With demand-push the cost is rather that for every proxy ‘cached’ $k(w/s)$ push messages will be sent along the edge where w/s is the number of writes per second (assuming k is expressed in seconds).

VII. THE SIMULATED TESTBED

We have produced a simulated testbed for examining properties of the push-pull framework in general and of given algorithms specifically. This testbed is a virtual-time based simulation of the framework which presents an interface (based on DSVs) to a virtual-time based model (the ‘input model’). The general system is described pictorially in fig. 2. The input model defines the set of simulation nodes, the set of variables and the ownership of variables by nodes. The framework simulation models the communication that takes place between nodes given a) a set of read/write operations from the model and b) a particular algorithm for managing update state of the edges.

Fig. 2. Architecture of the Testbed

In our case the input model was a set of traces collected over 1000 virtual time ‘ticks’ of a simplistic stochastic multi-agent model. This model is a grid environment where each grid cell has a variable integer ‘value’ which changes periodically through time. Simple agents move around the grid sensing the values of the cells according to that agent’s visual range. The model uses the DSV interface provided by the push-pull infrastructure, hence grid cells change their values using an explicit *write* operation and agents read the value of a cell using an explicit *read* operation.

VIII. RESULTS

The traces generated by the process described in section VII produce a single read-write sequence for each edge. To test each algorithm the same sets of traces were input to the testbed a number of times, each time a different algorithm was used to manage the update-state of the edges.

Figures 4-7 compare the performance of the three different algorithms; push-processing; pull-processing; and demand-push with various values of k . The performance is represented in these graphs as samples of the number of messages sent along all edges during consecutive 50-tick periods.

In order to evaluate the performance of the demand-push algorithm some understanding of its probable behaviour is necessary. Messages are generated on an edge under demand-pushing in 4 situations:

- 1) When a read occurs on a proxy and $T = 0$.
- 2) When a write occurs on a master and $T > 0$.
- 3) When a master receives a pull request (at the start of the period k) and replies with the current value and a confirmation of the flip to push.
- 4) When T times out and the edge is flipped back to pull mode by the replicator node sending a notification to the owner node.

Given these properties, the performance in terms of total aggregate network activity of the demand-push algorithm is not simply dependant on the ratio of reads to writes, though this is clearly a factor.

In fig. 3 three different values of k are shown for the same input sequence. In both cases where $k < 15$, the edge is reset to pull mode before the next read can reset T to k . In effect the algorithm is thrashing because more messages are being used for the management of update-state than for the communication of modelling information. However, increasing k is not necessarily the best solution to this problem. This is demonstrated by the increase in messages where $k = 10$ in comparison to where $k = 5$. The thrashing issue is still present, but unused push messages are now increasing the total communication.

Because the cost of thrashing is a coefficient of the read-frequency r/s of a given sequence, its significance will reduce in proportion to the write frequency of the same sequence. For example, were there to be a write every 2 ticks in the above figures, $k = 5$ would out perform the other two values, despite thrashing. Likewise an increase in the read-frequency will increase significance of thrashing up to a point at which $r/s = k$ when it will disappear.

To relate these observations to the results, we used the two variables of aggregate read frequency and aggregate write frequency, producing four permutations.

In general the worst performer is one of the two pure push-or pull-processing algorithms, for high read frequencies pull-processing produces the most messages, for high write frequencies, push-processing. Also in general, for higher values of k , demand-push out performs both pure-processing forms. Although there is little difference between them, $k = 25$

Fig. 3. The behaviour of the demand-push algorithm for a single operation sequence with various values of k . Shaded areas of each graph denote periods of push-processing. The numerical annotations on each message denote the cause of the message, as given in the enumeration above.

generally performs better for higher ratios of reads/writes while the opposite is true for $k = 100$. This fits with the analysis as $k = 100$ is more likely to leave a scarcely-read variable in push mode, causing more wasted messages in a high write-frequency model.

The effects of thrashing are abundantly clear for $k = 10$ where the read frequency is low. In fig. 6, all other algorithms result in almost no messages, demonstrating how few data are actually being exchanged from a modelling perspective. Conversely, and as speculated above, where the read frequency is high this thrashing component is not present, and $k = 10$ has a very similar profile to larger values of k . In fact it can be seen to out-perform both the other values at many points in fig. 7, where the write frequency is also high.

IX. RELATED WORK

As mentioned in section V, the push-pull framework in general is inspired by the PDES-MAS (Parallel and Discrete Simulation of Multi-Agent Systems) framework described by Logan and Theodoropoulos in [16]. This system also aims to achieve optimisation through bottom-up analysis of access

Fig. 4. Low read frequency/High write frequency

Fig. 5. High read frequency/Low write frequency

Fig. 6. Low read frequency/Low write frequency

Fig. 7. High read frequency/High write frequency

patterns. The framework in this case is a graph of owner nodes, each responsible for some set of shared-state variables. The primary aims are balancing of the computational load between these nodes, rather than minimisation of communication costs.

Work done by Yellin at IBM [17] presents a competitive algorithm for solving the problem of selecting a policy (in the form a component) for serving an application request based on previous performance profiles. This ‘delta’ algorithm is applied to the push-pull problem - termed the ‘pub/sub’ (publication/subscription) problem in Yellin’s work - and identified as optimal within the limit. However, Yellin’s approach does not explore the use of prediction models and we expect to find algorithms inspired by this area which outperform delta for a number of sequence types.

Vitter and Krishnan by comparison have done extensive work in the use of discrete sequence prediction models for improving prefetching strategies in virtual memory systems [18]. The modelling itself is directly applicable to the push-pull problem whilst its use to generate prefetching strategies is less so, due to the differences between push-pull and traditional caching mentioned in section VI.

X. SUMMARY AND FUTURE WORK

This paper has proposed a bottom-up approach to Interest Management in DVEs. Our work is currently in a continuing experimental stage with the essential machinery defined and proof-of-concept simulations completed.

Of most immediate interest is improving on the performance of the demand-push algorithm given in section VI. This algorithm is inelegant from the perspective of pattern-prediction and is susceptible to non-linear behaviour as shown in section VIII, making it a poor choice for robust systems. Mature research exists in the area of discrete sequence prediction and we propose to use this research in the development of more elegant, robust algorithms.

Implementing range queries to support an associative memory access model is another priority. A general technique for this support is given in section V-B. Expanding on this support

to include prediction of associative access sequences is a much more demanding challenge than that for reference-space access sequences.

Currently this work deals purely with real time applications. Future work will extend this to enable the framework to perform IM in as part of a larger virtual-time simulation framework. Mehl and Hammes work into implementing DSVs for virtual-time simulation systems [15] is a very useful starting point for this considerable task, particularly as it speculates on performance profiles for each architecture given properties of the model.

REFERENCES

- [1] K. Morse, “An adaptive, distributed algorithm for interest management,” Ph.D. dissertation, University of California, Irvine, 2000.
- [2] M. Macedonia, M. Zyda, D. Pratt, D. Brutzmann, and P. Barham, “Exploiting reality with multicast groups: A network architecture for large-scale virtual environments,” *IEEE Computer Graphics and Applications*, vol. 15, no. 3, pp. 38–45, 1995.
- [3] D. Schmalstieg and M. Gervautz, “Demand-driven geometry transmission for distributed virtual environments,” in *Proceedings of EUROGRAPHICS 1996*, vol. 15, no. 3, 1996, pp. 421–433.
- [4] G. Morgan and F. Lu, “Predictive interest management: An approach to managing message dissemination for distributed virtual environments,” in *Proceedings of the First International Workshop on Interactive Rich Media Content Production: Architectures, Technologies, Applications, Tools (Richmedia2003)*, 2003.
- [5] J. Barrus, R. Waters, and D. Anderson, “Locales: Supporting large multi-user virtual environments,” *IEEE Computer Graphics and Applications*, vol. 16, no. 6, pp. 50–57, November 1996.
- [6] T. Sweeney, “The Unreal networking architecture,” World Wide Web, <http://unreal.epicgames.com/Network.htm>, 1999.
- [7] GarageGames, Inc., “openTNL - torque network library,” World Wide Web, <http://www.opentnl.org>, 2004.
- [8] IEEE, “1278: Standard for distributed interactive simulation,” 1995.
- [9] H. Abrams, K. Watson, and M. Zyda, “Three-tiered interest management for large-scale virtual environments,” in *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, 1998, pp. 125–129.
- [10] J. Sorroche and J. Szulinski, “Bandwidth reduction techniques used in DIS exercises,” in *Proceedings of the 2004 European Simulation Interoperability Workshop*, 2004.
- [11] US Defence Modelling and Simulation Office, “HLA Interface Specification, version 1.3,” 1998.
- [12] K. Morse and J. Steinman, “Data distribution management in the HLA: Multidimensional regions and physically correct filtering,” in *Proceedings of the 1997 Spring Simulation Interoperability Workshop*, 1997.
- [13] C. Faisstnauer, D. Schmalstieg, and W. Purgathofer, “Scheduling for very large virtual environments and networked games using visibility and priorities,” in *Fourth IEEE International Workshop on Distributed Simulation and Real-Time Applications (DS-RT’00)*. IEEE, 2000.
- [14] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*. Wiley Interscience, 2000.
- [15] H. Mehl and S. Hammes, “Shared variables in distributed simulation,” in *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, 1993, pp. 68–75.
- [16] B. Logan and G. Theodoropoulos, “The Distributed Simulation of Multi-Agent Systems,” in *Proceedings of the IEEE - Special Issue on Agent-Oriented Software Approaches in Distributed Modelling and Simulation*, 2000.
- [17] D. M. Yellin, “Competitive algorithms for the dynamic selection of component implementations,” *IBM Systems Journal*, vol. 42, no. 1, pp. 85–97, 2003.
- [18] J. S. Vitter and P. Krishnan, “Optimal prefetching via data compression,” *Journal of the ACM*, vol. 43, no. 5, pp. 771–793, September 1996.