HLA Proxy: Towards Connecting Agents To Virtual Environments by Means of High Level Architecture (HLA)

Tomas Plch[1], Tomas Jedlička[2], Cyril Brom[3]

Faculty of Mathematics and Physics, Charles University in Prague
[1]tomas.plch@gmail.com [2]jedlickat@gmail.com [3]brom@ksvi.mff.cuni.cz

Abstract. Coupling virtual environments (e.g. game engines like Source Engine or Unreal Engine 3) with agent reasoning systems (ARS) is often used in the multi-agent systems (MAS) research field. However, externally connecting ARS or MAS to environments almost always requires individual approach for every coupling. Therefore, we recognize the need for a common method of access, without the need to implement a network stack, network protocol or data management. In this paper, we present our new project - HLA Proxy - utilizing the High Level Architecture (HLA) standard (IEEE 1516-2010) for interconnecting simulations and simulators. We created a C++ prototype middleware providing universal and transparent access to the HLA infrastructure for not HLA-capable applications (i.e. ARS, MAS, visualization tools etc.), thus allowing cross-platform, distributed connection to environments and between environments. Our work is aimed at being directly integrated into the environment (i.e. engine) and application via dynamic linkage. Here, we present our architecture and our proof-of-concept integration into CryENGINE 3 (used for the Crysis game) and Source Engine (used for the HalfLife 2 game) running on Windows XP 32bit and Windows 7 64bit platforms. We also implemented a 64bit Linux console application utilizing HLA Proxy to connect to both engines capable to send console commands and receive environment updates.

Keywords: HLA, High Level Architecture, middleware, Agent Reasoning System, Computer Games, Distributed simulation, Dynamic-Link Library

1 Introduction

The realism of virtual environments (e.g. computer games) increases with every iteration of their respective engines (e.g. Source engine [1], CryENGINE 3 [2], Unreal engine [3]). These worlds being extensively realistic are excellent candidates for conducting research and experiments in various fields of Artificial Intelligence (AI) [18], ranging from crowd simulations to single agent reasoning systems (ARS) and multi agent systems (MAS). However, it is fairly complicated to access these virtual worlds, possibly by simple, universal and versatile means.

The CIGA middleware [24] represents an attempt to conceptualize a general architecture needed for coupling computer games with ARS and MAS. CIGA presents a layered fairly complex architecture for accessing game engines, comprised of a physical, semantic and cognitive layer. It is build around the notion of *ontology domain*, which is used to unify the environment's and the agent's view of semantic representations. However, CIGA's physical layer only encapsulates the problem of coupling to an environment. Our opinion is that CIGA creates conceptual and run-time overhead over the coupling, which should be focused on universal data exchange. Actual data interpretation, abstraction and management should be located within the interconnected entities (i.e. MAS, ARS, game engines etc.).

We distinguish the following methods for accessing virtual environment's data (e.g. objects, events etc.) and functionality (e.g. actions of agents): 1) direct access and 2) via external interfaces. Direct access is facilitated via including compiled or scripted code into the engine's runtime, either by dynamic linkage (e.g. Gary's Mod [8]) or loading and executing scripts (e.g. utilizing LUA scripting language [5]). Access based on reverse engineering is rare (e.g. StarCraft Brood War API [7]).

Access over an external interface is commonly realized via network sockets utilizing a text-based or binary protocol. External coupling to an engine requires a bidirectional (network) interface. Often, a custom interface is designed (i.e. network stack, network protocol, data update management, etc.) based on the respective environment's architecture [4] (e.g. event/object based architectures, single/multithread internals etc.). Based on our experience with Source Engine, Cry Engine, Unreal Engine, and Defcon game [13], we consider this task a significant time-consuming effort.

However, need to utilize a 3D virtual environment lead developers and researchers to create an external network interface for Unreal Engine. The resulting text-based GameBots [9] protocol is used to export data and events utilizing the UnrealScript scripting language. The protocol is utilized by many researchers due to the fact it provides an existing network interface for coupling to a capable virtual world.

The Pogamut project [10] capitalizes and extends the idea of GameBots and incorporates the protocol via the GaviaLib library into a NetBeans based platform (Figure 1) for prototyping IVAs utilizing the Java programming language [11]. Presently, Pogamut is limited to few environments – a) Unreal Tournament 2004, b) Unreal Development Kit based projects – e.g. the Emohawk project [12], and c) Defcon game [13].

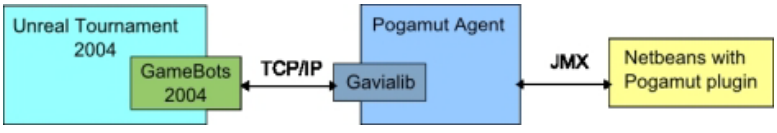

Fig. 1. Pogamut project architecture – connecting a Pogamut Agent via GaviaLib and GameBots 2004 to the Unreal Tournament 2004 computer game. The agent is controlled by a decision making mechanism running in the Netbeans development environment and is written using the Java programming language

Due to the fact, that coupling methods between applications, ARS, MAS and virtual environments are limited to used protocols (e.g. GameBots) and architectures (e.g. CIGA), we recognize the need for a more universal, platform independent (i.e. Linux/Windows 32/64bit), direct (i.e. without various layers like presented [24]), adaptable and fast solution, which can be easily integrated into most environments or projects with less work on the network and data management overhead currently involved.

This paper presents our C++ based platform *HLA Proxy* for universal, direct, adaptable, cross-platform interconnecting of applications – e.g. simulations, decision making mechanisms, data collecting applications, ARS, MAS etc. HLA Proxy exploits the High Level Architecture (HLA) standard [6] for interconnecting simulations and simulators. Our middleware is aimed at providing the HLA capability to most applications build upon the object oriented programming paradigm, by integrating our middleware via dynamic or static linkage. The target application can utilize a subset of the HLA standard's capabilities and can exchange data in an object or event driven way without the need to implement a network interface or a translation mechanism between the inner representation and a network protocol stack. Our architecture, in contrast with CIGA [24] is aimed at mitigating data between the virtual environment and the ARS or MAS via HLA.

High Level Architecture has already been used for virtual agent's research by (Lees et al.) [25]. HLA compliant agents created with the *SIM_AGENT* high level design toolkit [26] were introduced to a tile world scenario and compared with native SIM_AGENT's agents and their performance was inspected. HLA compliant agents performed worse than native SIM_AGENTs in the presented scenario. However we think this is mostly due to the tendency of the scenario and the HLA agent's design. The low performance of the agent's is mostly due to the overuse of the HLA's synchronization mechanisms. Also the integration with SIM_AGENT toolkit might represent a bottleneck responsible for the degraded HLA compliant agent's performance. However, a degraded performance of HLA in respect to an optimized engine-ARS coupling is expected, because HLA and HLA Proxy are aimed at providing *universal* access to environments and simulations/simulators within large simulation aggregations which requires various non trivial time consuming mechanisms to be present (e.g. time management, data delivery systems etc.).

Note that, the LVC Game proprietal solution by Calitrix [15] provides a similar solution as the HLA Proxy. It provides a network layer for various applications (e.g. Virtual Battle Space 2 (VBS2) [14]) implementing a subset of the DIS/HLA standard. However, LVC Game supports only a limited subset of the military RPR Federation Object Model 2.0 [17]. Therefore, it is not feasible for AI research, because the needs of decision making mechanisms for IVA's are broader then the representation used.

The paper is structured as follows – the following section is focused on presenting basic High Level Architecture concepts. Section 3 is aimed at presenting our middleware. Section 4 and 5 is focused on our proof-of-concept implementation and performance tests of HLA Proxy's internal database. Section 6 concludes and presents future work.

2 High Level Architecture

The purpose of this section is to provide insight into basic High Level Architecture (HLA) concepts and ideas. The standard [6], [16], [21] itself was created for the purpose of interconnecting a multitude of various simulations and simulators in use by the United States Department of Defense [18] without expensive new development or redesign of current simulations and simulators (e.g. life-size tank simulator). HLA recognizes the term *federation*, which represents the aggregation of participants – *federates* (Figure. 2). A federate can be perceived as any application, simulation or any other entity, passively or actively participating in the federation. In our case, federates are either virtual environments or decision making mechanisms.

A federation setup example could be as follows – a virtual fighter jet simulator (e.g. Lock On: Modern Air Combat [19]) being one federate having an AI controlled fighter jet in the simulation, a soldier simulator (e.g. Virtual Battle Space 2 [14]) being operated by a human and a human crew operated tank simulator being the third federate. All these participate and coexist in one simulation environment, where they have different internal data representations and implementations, and care about various degrees of data abstraction – e.g. the fighter jet simulator does not need to know how much health the soldier has, or how the physical model of the tank works. Furthermore, if a virtual soldier mounts a tank as a driver, the soldier simulator is responsible for the abstraction and virtual presentation of the real tank simulator. The data exchanges and communication between federates is done in the HLA environment, where the HLA's mechanisms are responsible for correct exchange and delivery of data to all participants.

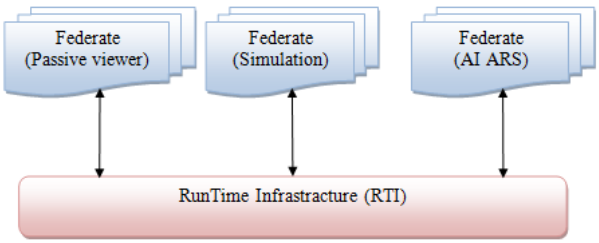

Fig. 2. High Level Architecture – a passive federate (e.g. mission logger), an active federates (e.g. tank simulator and virtual entity managed by an AI)

2.1 Data representation and exchange

Interconnecting various federates requires the federation to share a common view of the simulated world, at least conceptually. HLA specifies this common view as the *Federation Object Model* (FOM) [16], which is similar to the domain ontology [24]. The FOM is a tree-like structure based on the object oriented representation of the world (Figure 3) and is represented by a XML document. Nodes within the structure are called O*bject Classes*, where the descendant inherits the parent's attributes. The concept allows for backward compatibility, where when adding a new federate with a

more detailed notion of the world (i.e. more deep FOM structure) can work with older federates who recognize only a portion of the updated FOM. The HLA standard also allows for complex type creation by type aggregation.

The proper exchange of data within a federation is facilitated via the Run-Time Interface (RTI). The RTI represents the actual implementation of the data exchange protocols. The data exchange between federates is based on ownership, update status and time management. The RTI also provides a multitude of services [21] – e.g. ownership acquisition, object discovery etc.

The data exchanged can be of two major types a) *Object Instances* and b) *Interactions*. The Object Instances represent the object of a certain Object Class and their attributes within a simulation world (e.g. a virtual soldier) specified in the FOM. The attributes and Object Instances can be owned by a particular federate which is responsible for their update and other federates can subscribe to these updates. Parameterized Interactions can be seen as events that occur in the simulated world (e.g. a grenade explosion). The data exchange model is a *publisher/subscriber model*, where federates publish and subscribe to Object Instances and attributes. The data update model is a *one writer, many readers model*, where ownership is acquired or relinquished over RTI.

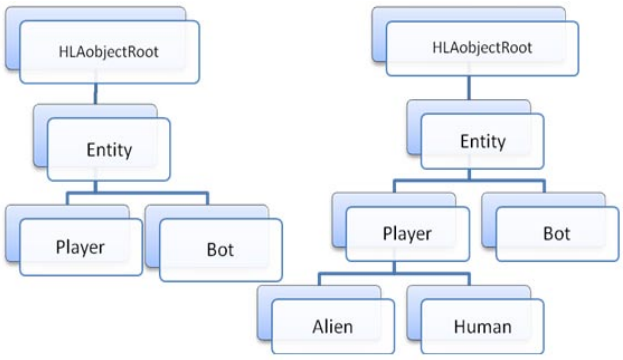

Fig. 3. Federation Object Model representation of two ontological domains. Both domains represent an example of how entities like Players and Bots [4] can be represented. The domain on the right is an extension of the left domain, where the Player objects class is a parent of the Alien and Human object classes.

2.2 Example

To better illustrate HLA's workings, we provide a simple example of a federation's data flow where a computer game engine is coupled with an ARS. Let us assume a virtual world where secret agent bots live and are capable of shooting at each other. The virtual world is run by a computer game engine (e.g. Source Engine) and the ARS is a simple C++ application with simple reactive reasoning. The engine is a federate and every secret agent bot has one dedicated ARS federate. The FOM of this

federation specifies only the *Agent* Object Class with a *Health* attribute and one *Attack* Interaction with two parameters – "who attacks who".

The instance of the *Agent* Object Class called *Agent007* is owned by its creator, the Source Engine. The ARS being situated as different federate, can subscribe to some attributes of *Agent007* – e.g. his health, position and enemy agents it can see. Let us assume that, *Agent007* meets an enemy agent called *Agent001* (also an *Agent* Object Class).

The notion of *meeting of two agents* can either reasoned by the agents themselves, based on percepts from the environment (e.g. actual percepts in the agent's field of view, or computations based on the knowledge of all agent's positions). However, this inference can be done by another federate responsible for determining visibility between agents and propagated as an HLA Interaction to the federation. A similar HLA Interaction could also originate from the engine. All variants are equivalent, the difference is only where the information about visibility is processed.

Let us assume that Agent007's ARS reasons the need to shoot Agent001 and it triggers the *Attack*(Agent007, Agent001) HLA Interaction. The Interaction is delivered to the engine (if it has subscribed to receive it) over RTI. Agent007 shoots and may trigger or update various attributes within the engine (e.g. health count of Agent001, ammo count in Agent007's pistol etc.).

It is noteworthy that both Agent001's and Agent007's ARS can have various internal representation's of the ontology domain, even various degrees of perception. Agent007 might not know about having a pistol, only knowing the interaction of *Attack* which is handled engine specific – by pistol, (in a different game) by bow or not at all (i.e. Agent007 has no weapon or no ammo).

3 HLA Proxy Middleware

This section is focused on explaining the basic features of our HLA Proxy middleware, our design goals and decisions. The main idea behind our middleware is to provide any object-based application or engine with the capability to access the HLA and to allow data exchange without being limited to one protocol or architectural design. The HLA Proxy's philosophy is to only load the library into the host's run-time, couple the internal objects to objects represented in the FOM and perform write and read operations on those Object Instances. HLA Proxy should handle all the synchronizing of data between federates, to keep everybody up to date.

3.1 Design

The main design issues we addressed with our HLA Proxy were a) *generality* – virtual environment or application capable of dynamic/static linkage should be capable to use our middleware, b) *adaptability* – if the view of the world or requirements on functionality change, we have to be able to reflect it, c) *transparency & simplicity* - we cannot bother applications or engines with handling network traffic

or data management, d) *responsiveness* – the middleware has to be fast to be able to cope with engines.

Our approach is based on the notion of hiding the network management, object management, semantic transformations etc. from the user (Figure 4) thus achieving *simplicity*. We consider the best option to directly couple engine's objects, semantic transformations, and inferences etc., to Object Classes and Interactions specified in the FOM.

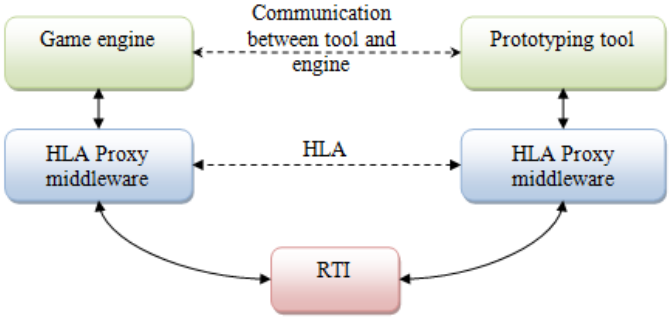

Fig. 4. Coupling of a Protoyping tool (e.g. Pogamut) to a Game Engine over the HLA Proxy middleware over the RTI using HLA. The Game Engine/Prototyping tool integrates the HLA Proxy middleware via dll linkage and accesses proxy classes generated based on the FOM by get/set methods. The updates are propagated over the RTI between the HLA Proxy nodes and deliver the information to either the Game Engine or Prototyping tool.

The federation's ontology (represented within the FOM specification) can be either taken from already existing FOMs (e.g. RPR FOM 2.0) or developed on a per-case basis as a common derivate of the ontology and abstractions of the participating federates. To our experience, the best practice is to take the existing exported object declarations (e.g. C++ headers of the engine's SDK) and build the FOM based on them. This allows for fast and simple integration of the resulting HLA Proxy's mechanisms into the present engine's or application's code (e.g. the *Player* object instance within the engine calls update functions on the FOM's *Player* class that reflect the actual *Player* Object Instance in the federation and map to a *HumanPlayer* object in another federate that is coupled with the FOM's *Player* Object Class).

The HLA Proxy provides the means to mitigate the ontology over the HLA environment by invoking Interactions or updating and requesting the data on Object Instances from our middleware (Code 1). The update methods of HLA Proxy can be either called directly from an engine code segment (e.g. *Player::setHealth()* method), engine callbacks, engine update loop, or within a code segment present in an additional layer developed for the coupling. This allows for more complex, better tailored integration into an engine or application. Conceptually, this approach allows for a cleaner and less bounding integration in respect to the engine – i.e. the updates can be per objects, or at specific code locations etc (e.g. more important objects are updated every frame, less important objects are updated during update loops).

```
class CPlayer{/* class declaration within CryENGINE 3 */
      /* usual members for engine purposes start here */
      unsigned int m_health;
        ...
      /* HLA Proxy code starts here */
      HLAProxy::Data::Player *pHLAPlayer;
};

CPlayer::setHealth (unsigned int hp) {
    /* engine specific code goes here */
    m_health = hp;
 /* HLA specific code for propagating updates outwards */
  if (pHLAPlayer != NULL) {
    pHLAPlayer->setHealth(m_health);
    pHLA->updateInstance(pHLAPlayer);
  }
}
```

Code 1. Example of a C++ code from the Cry ENGINE 3 SDK, where the CPlayer is an object within the Cry ENGINE 3 environment representing the player's embodiment. When the engine updates the *health* of the player, the HLA Proxy middleware is called to propagate the update to other federates

3.2 Internal Architecture

To satisfy *generality*, we designed HLA Proxy as a dynamically linkable library (*dll* in Windows based systems, *so* in Linux based systems), which can be introduced into the environment's or application's runtime.

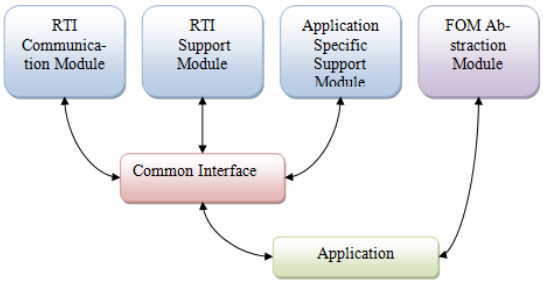

Fig. 5. Barebone HLA Proxy two tier architecture – components are presented in boxes, lines represent communication interfaces between components. The Application box represents the run-time dynamically loading HLA Proxy.

It can be seen in Figure 5, HLA Proxy's architecture is a simple two tier architecture. Application or engine can access the HLA Proxy by two means – the

Common Interface and the *FOM Abstraction Module*. Common Interface provides basic functionalities, like startup, shutdown, registering handlers for HLA Interactions. The other modules, like *RTI Support Module* and *RTI Communication Module* provide support for RTI related network transfer and can be specific or optimized per RTI to gain a performance boost.

The FOM Abstraction Module represents the means for the application to access the federation's ontology domain specified within the FOM XML document. To satisfy the goals *transparency*, *simplicity* and *adaptability* we needed to address the following issues: 1) extract the ontology domain from the FOM module (satisfies adaptability), 2) provide means for access to the data via simple means (satisfies simplicity), 3) mitigate data exchange from and to HLA in respect to correct data updates (satisfies transparency).

First, we needed to extract the ontology domain specified in the FOM. Due to the fact that multitude FOMs exists for various uses (e.g. RPR FOM 2.0 for military simulations) and the FOM can change, update or be replaced, it is feasible to assume that hard-coding a FOM into HLA Proxy was not a suitable solution. We chose to provide our middleware with the capability to *generate C++ code* based on the FOM XML specification using our XSLT code developed for this purpose. Because the FOM's structure is derived from the object oriented paradigm, the generated code can reflects the ontology by C++ classes in an inheritance schema equivalent to the FOM's specification.

Second, we needed to access the in the FOM specified Object Classes, their attributes as well as Interactions by simple means. Invocation of an Interaction can be done easily by calling the appropriate *global* function from the host (i.e. application, engine) run-time. The host run-time can register callbacks from HLA Proxy to be called when an Interaction occurs within the federation. Objects Instances as being a result of creating an Object Instance (stored in a in-memory database within HLA Proxy) and can be accessed via *Handlers* which are initialized from an internal hash table based on the unique Object Instance names.

To access object attributes, we utilize the *get* and *set* approach, where every attribute has his own *get/set functions* generated (i.e. FOM attribute *Foo* has functions *getFoo*() and *setFoo*()). The *get* and *set* functions are generated based on the access specification of the attribute – i.e. *publish-only attributes* have only *set* functions, *subscribe-only* have only *get* functions and publish/subscribe have both. An example of use can be seen in the code above (Code 1). It is noteworthy, that the attribute *Foo* might represent one attribute within one federate's internal object and a combination of attributes in a different federate's internal representation. Every federate is responsible for its own interpretation of the FOM.

Third, we needed to provide means to synchronize the host run-time with the access to the distributed environment of HLA. On one side, operation requests are inserted by the host run-time, from the other side, updates are received via RTI. We designed our own in-memory internal database of Object Instance Attributes, mostly due to the fact, that available database solutions (i.e. databases like MySQL, PosgreSQL etc.) are either standalone or not suitable for our specific requirements. We need to keep transactions *isolated*, but cannot perform a *rollback* or transaction

aborts, because virtual worlds tend not to be able to rollback. We also have to keep the data *consistent* and *ordered by timestamps*. We use a combination of locking, assigning unique timestamps to requests and multiversion approach [22]. The internals of our database design are beyond the scope of this paper. As for *responsiveness*, we designed the HLA Proxy's database internals to be *fully multi-threaded*. Internally we use one *dispatching* thread and an army of *worker threads* which execute the request on top of the database.

The resulting FOM Abstraction Module encapsulates all the functionality for the host run-time to access FOM specified federation's ontology domain over RTI in *adaptive*, *simple* and *transparent* way. Our in-memory database provides the host environment with accurate data in respect to the host's run-time HLA time management specification (i.e. update orderings depend on this). The multi-thread design allows for parallel operations on attributes, thus providing more efficient use of today's multicore hardware.

4 Proof-of-concept implementation

The aim of our proof-of-concept is see if our approach is feasible and working by exchange information (e.g. health status and command scripts) back and forth between federates (i.e. computer game and application). We also aim at integrating HLA Proxy into two major computer game engines with different internal architecture – Source Engine (Half Life 2 game) and Cry ENGINE 3 (Crysis game). The integration via dynamically loading our dll was performed without creating additional layers of architecture to the engine (i.e. network layer, abstraction transformation etc.).

We created a federation between our console application and each of the game engines (separately) – i.e. the federation had only 2 federates (due to our license for the MÄK RTI [17]). Both engines were running on a Windows XP 32bit and Windows 7 64bit platform and our console application was developed for a Linux 64bit platform (Figure 6).

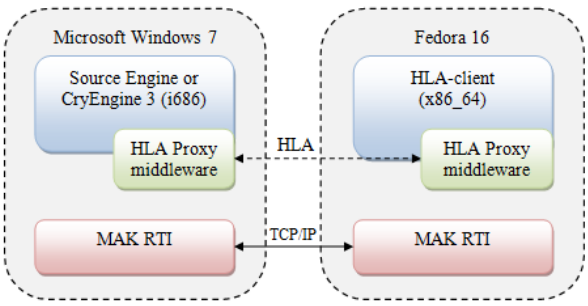

Fig. 6. HLA Proxy proof-of-concept setup where Source Engine/CryEngine 3 is talking over HLA Proxy with a client application over the MÄK RTI working over TCP/IP network

We considered a simple scenario where our console application receives updates on health status of a Non Player Character (NPC) and sends script commands to the environment (which would appear in the respective engine's console) via HLA Proxy. In our scenario, we forced the NPC via script command to throw a grenade and kill himself, thus receiving health updates before and after the explosion and receiving Interaction notification of the explosion.

The integration into Source engine took us about 9 months of work, due to the fact that the integration was actual HLA Proxy's design and development. We used the Source Engine as a design reference due to its complex internal mechanisms. Our additional findings in respect to integration issues (e.g. memory management issues in Half Life 2 etc.) can be found in [23]. The integration into CryENGINE 3 was rather quicker, due to our experience with Source Engine – it took about 3-4 days.

5 Performance

This section is focused on providing results of our performance benchmarking of the HLA Proxy middleware. We conducted a series of synthetic tests on our in-memory database to inspect the scaling properties of our scheduling algorithm and data storage mechanism. We decided to only benchmark the in-memory database for two reasons – 1) we had no complex enough federation at disposal and 2) a federation wide testing would be about benchmarking the used RTI (i.e. it would depend on the federation properties and network topology).

We base our hypothesis on the observation that frame-rates of game engines is presently around 50–60 frames per second at most. Our expectation would be the database's capability to handle 100 operations (i.e. read/write operations) per frame to satisfy a reasonable assumption on how many objects interact or change during a single frame. It is noteworthy that our expectations are not based on actual measurements, because the amount of interacting objects can vary based on engine and in-game situation.

5.1 Benchmarking method

Our benchmarking is focused on the duration of read and write operations in a barebone setup of HLA Proxy to avoid resource consumption by other modules (e.g. logging etc.). The HLA Proxy was integrated into an application accessing Object Instances and their attributes. We accessed *one* instance of a HLA Object Instance, because internally we either use direct access over directly linked *handlers*, or a hash table. In most use cases, the handler would be linked once during in-engine object creation and then reused when accessed or updated. We also did not want to benchmark our hash table implementation, but our scheduling mechanism. It is noteworthy, that our database works with Object Instance Attributes, rather than whole Objects Instances, therefore needed only one instance of an Object Instance due to the fact that only concurrent access to Object Instance Attributes are of interest to us – they cause the actual slowdown when scheduling operations. Access to

multiple different Object Instances is handled in parallel. It is noteworthy that the in-memory database runs in parallel with the application's code, where one dispatcher thread is responsible for 10 worker threads that perform the requested operations.

We established four use cases for read/write operation ordering:

- series of reads,
- series of writes,
- series of writes followed by a reads,
- random ordering of reads and writes.

For every use-case we perform a series of tests, where we access the attributes in every iteration in the following setups:

1. single attribute from one single thread,
2. multiple (4) attributes accessed from one single thread at once (all in one iteration),
3. multiple (4) attributes access from 4 threads (one attribute per thread).

The iteration count for a batch of tests is increased from 100 to 20000 iterations per run. We conducted 5 runs for every combination of setups and the resulting value is a mean of the measured runs.

Resulting time of executions are normalized to operations processed per second. This representation can easily show whether such performance meets expectations for real-time usage or not. Measurements contain not only time consumed by processing of requests but also time spent in scheduling of request in application code.

We performed our evaluation on a Windows 7 64bit operating system running on a Core 2 Quad 2.4GHz (E6600) processor. Our build target was a 32bit platform. The actual testing application was running on a Virtual Box hosted Fedora 16 platform with one dedicated CPU for the virtualization.

5.2 Results

In Figure 7 we show the read throughput of the database. Because read operations are blocking operations (i.e. have to finish before returning to the caller), we do not need to perform Setup 2, because it behaves like Setup 1. The scheduling mechanism scales properly in both the single and multithreaded setup.

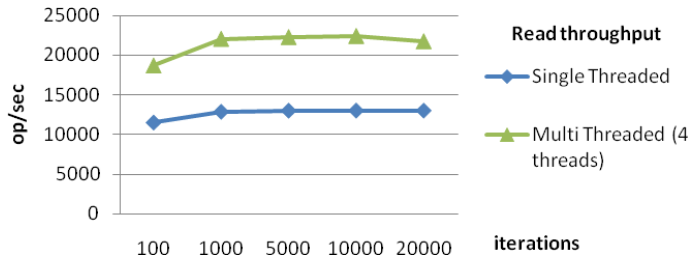

Fig. 7. Read throughput – attributes accessed by long series of reads

In Figure 8 we show the write throughput of the database. The performance for Setup 1 and 2 perform as expected. In Setup 2, write operations are asynchronous and can be performed in parallel - the database works with Object Instance attributes, rather than whole objects. Therefore the increase in Setup 2 in respect to Setup 1 is almost four times. The Setup 3 behaves slightly better, because the inserting into dispatcher's request queues is done on behalf of the requesting thread's runtime and therefore can insert more requests to be processed by the dispatcher thread.

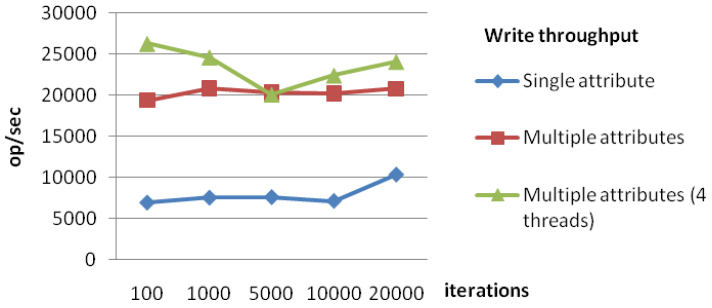

Fig. 8. Write throughput – attributes are access in a long series of writes

In Figure 9 we show a series of writes followed by a read. This ordering of operations is interesting due to the fact that the read has to wait until all the writes are processed to acquire the current data. All setups can be seen to scale well. Setup 1 and 2 behave as expected – write operations are processed asynchronously and only final read operations do block. Parallel processing of asynchronous writes in Setup 2 might provide the observed speedup. The Setup 3 also behaves as expected and scales well. The doubling in speed is due to the same effect described earlier.

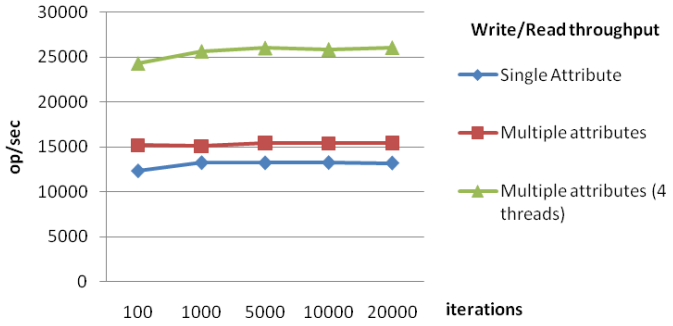

Fig. 9. A series of writes followed by a read operation

In Figure 10 we can see a completely randomized ordering of requests for single thread and multiple thread setups. Both setups behave as expected – they come close to each other because of the necessary synchronization on attribute asynchronous writes and blocking reads which have to wait for each other. The multi-thread setup is

still better then the single thread setup, but both are usable, thus HLA Proxy does not favor a single or multi thread design approach.

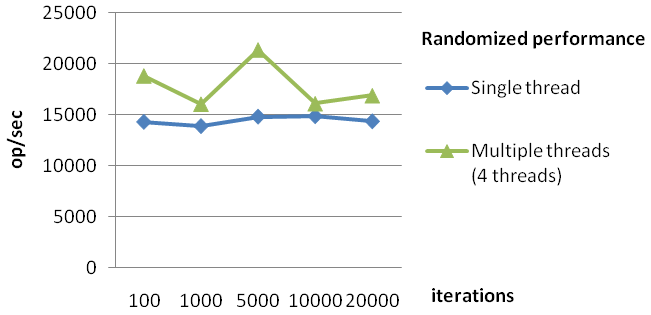

Fig. 10. Randomized ordering of requests on read and write operations

5.3 Discussion

Performance results collected by our benchmark look promising. The middle value of performance equals to 16860.6 (multithreaded) and 14285.7 (single-threaded) random operations per second. Most engines are limiting its frame rate to 50-60 frames per second. Therefore the database can perform around 250 operations per single frame. This result is beyond our initial expectation of 100 operations per single frame. Engines could adapt to the current load of updates by scheduling less operations thus providing a less accurate virtual world's representation. To conclude, the performance meets expectations for usage of the middleware for interconnection between computer game engines and ARS or MAS.

One limitation is that maximum possible performance is limited. Measured limits can be seen in Figure 7 and Figure 8 and are close to 25000 operations per second. The performance bottleneck is the thread scheduling mechanism and it does not matter how many cores the machine has, because we use only one dispatcher thread.

Due to the fact, that most operations on an object's attributes within the engine are performed within one thread's runtime – the most important information is how fast HLA Proxy is processing operation requests from a single thread. A performance improvement might be achieved by using multiple scheduling modules, thus introducing more dispatcher threads for single non-concurrent thread access on different attributes. Unfortunately such modification would introduce a more complex scheduling algorithm and thread synchronization issues.

Due to the fact that the HLA Proxy middleware would run within an engine's runtime it is expected that performance of the internal database will degrade, because game engines tend to consume enormous resources – i.e. memory and processor time. Engine's resource consumption is beyond our reach and this issue has to be addressed on a per integration basis.

6 Conclusion and Future Work

In this paper, we presented the C++ based HLA Proxy middleware prototype for interconnecting applications and engines, as well as various engines with each other in a general, simple, transparent and accessible way. We developed an architecture with good results in synthetic benchmarking in respect to internal operation.

Our proof-of-concept integration with CryENGINE 3 and Source Engine proved to be successful, despite the fact that the SDKs provided are undocumented. Our middleware is aimed at shortening the time for creating an application↔game engine coupling – HLA proxy hides network management and data management layers from the developer and provides simplistic update mechanisms (e.g. simple function calls on *get*, *set* update functions, Interaction notification callbacks etc.). We also managed to connect to CryENGINE 3 in less than a week.

The use of HLA Proxy is simplistic and can be integrated into an application via dynamic or static linkage. The capability to generate code (i.e. get and set update function etc.) based on the FOM provides a unique solution not only to connecting to environments, but also when exchanging data in various domains.

For future work, we need to connect a true decision making mechanism and a development platform to at least one environment. We also look forward to enhance the capabilities of HLA Proxy to support more of the current HLA standard – namely various time management mechanisms. We also plan to test our architecture in a more complex environment with multiple simulations – create a federation where two different engines are connected (e.g. Source engine and CryENGINE 3) to an agent decision making mechanism. We also intend to study integration times and capabilities for HLA Proxy and Pogamut/GameBots integration into a computer game engine.

Acknowledgments: This work was partially supported by SVV Project no. 265 314, student grant GA UK No. 0449/2010/A-INF/MFF, by project P103/10/1287 (GA ČR) and GA UK No. 655012/2012/A-INF/MFF.

1. Valve: Source SDK (2011) [url: http://source.valvesoftware.com/sourcesdk.php 20.2.2012]
2. Crytek: CryENGINE 3 SDK (2011) [url: http://mycryengine.com/ 20.2.2012]
3. Epic Games: Unreal Tournament 2004 (2004), [url: http://www.unreal.com (20.2.2012)]
4. Gemrot, J., Brom, C., Plch, T.: A periphery of Pogamut: from bots to agents and back again. In: Agents for Games and Simulations II, Springer (2011).
5. Ierusalimschy, R., Celes. W., de Figueiredo L., H.: Lua programming language [url: http://www.lua.org/ 20.2.2012]
6. IEEE1516.1-2010 IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Federate Interface Specification (2010)
7. BWAPI (2004) [url: http://code.google.com/p/bwapi/ 20.2.2012]
8. Facepunch Studios: Garry's mod (2004) [http://garrysmod.com/ 20.2.2012]
9. Adobbati, R., Marshall, A., N., Scholer, A., Tejada, S., Kaminka, G., Schaffer, S., Sollitto, Ch.: Gamebots: A 3d virtual world test-bed for multi-agent research, Proceedings of the 2nd international workshop on Infrastructure for Agents MAS and Scalable MAS (2001)

10. Kadlec, R., Gemrot, J., Bída, M., Burkert, O., Havlíček, J., Zemčák, L., Pibil, R., Vansa, R., Brom, C.: Extensions and applications of Pogamut 3 platform. In: Proc. 9th IVA, Springer (2009)
11. Gemrot, J., Brom, C., Bryson, J., Bída, M.: How to compare usability of techniques for the specification of virtual agents' behavior? An experimental pilot study with human subjects. In: Proceedings of Agents for Games and Simulations, AAMAS workshop (2011)
12. Bida, M., Brom, C.: Emohawk: Learning Virtual Characters by Doing. In: Proceeding of ICIDS 2010, Springer, (2010)
13. Introversion Software: DEFCON (2006) [url: http://www.introversion.co.uk/defcon/ 20.2.2012]
14. Bohemia Interactive: Virtual Battle Space 2 [url: http://vbs2.com 20.2.2012]
15. Calytrix Technologies: LVC Game
16. IEEE 1516.2-2010 Modeling and Simulation (M&S) High Level Architecture (HLA) - Object Model Tempalte (OMT) Specification
17. VT MÄK, [url: http://www.mak.com 20.2.2012]
18. Department of Defense, [url: http://www.defense.gov/ 20.2.2012]
19. van Oijen, J.; Dignum, F.; Scalable Perception for BDI-Agents Embodied in Virtual Environments. Web Intelligence and Intelligent Agent Technology (WI-IAT) (2011)
20. Eagle Dynamics: Lock On: Modern Air Combat (2003)
21. IEEE 1516 Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules (2010)
22. Garcia-Molina, H., Ullman, J., Widom, J.: Database Systems: The Complete Book, Prentice Hall (2001)
23. Jedlička, T.: Utilizing HLA for agent based development platforms. Master thesis, Charles University (2012)
24. Oijen, J. van; Vanhée, L. and Dignum. F.: CIGA: A Middleware for Intelligent Agents in Virtual Environments In Proceedings of the 3rd International Workshop on Agents for Education, Games and Simulations, AAMAS11 (2011)
25. Lees, M., Logan, B., Theodoropoulos, G.: Agents, games and HLA. Simulation Modelling Practice and Theory, (2006)
26. Sloman A., R. Poli, R., SIM_AGENT: A toolkit for exploring agent designs. Intelligent Agents II: Agent Theories Architectures and Languages (ATAL-95), Springer (1996)