Abstract—The area of virtual environments has received an increasingly growing attention from the research community, leading to the design of various innovative albeit incompatible protocols. In this paper, we present an open modular middleware that allows the clients and servers of multiple virtual worlds to be dynamically interfaced via a granular design. Our technology significantly alleviates the workload traditionally imposed to developers, and we see it as an important step forward towards the design of universal interfaces for interoperable virtual environments.

Keywords—protocols, virtual environments, interoperability, middleware

I. INTRODUCTION

Over the last two decades, a number of virtual environment systems have been developed, some of which have experienced either moderate or even large successes [1], whereas others have simply become deprecated [2]. Many of these were the first virtual environments of their kind and their developers proposed various innovative solutions in the areas of scalability, efficient communication, graphics, system architecture and other. The need to overcome remaining limitations in the number of concurrent avatars, visual detail, user freedom and scalability has sparked many interesting research projects across various research groups in the world [3], [4].

In these projects, however, solutions are often limited to one particular system or can only be integrated into one particular virtual world. Most of these systems define a fixed on-the-wire protocol that describes how bits are sent on the network. These protocols are often incompatible with each other, which makes it difficult to reuse the proposed solutions. Some of them define many high-level messages that are very specific to a particular virtual world system. Others, in an attempt to make the protocol more generic, define low-level messages only, but leave a lot of work to the developers of the virtual world.

In this paper, we propose a novel solution—an open modular middleware, which (i) allows to dynamically check which features are supported by the remote end, (ii) is designed to be extensible, (iii) can be used with existing systems and (iv) is simple to program.

We start by outlining existing solutions in Section II. The concepts behind our middleware design are then discussed in Section III and their implementation in Section IV. In order to evaluate the benefits of our approach, we present practical results in Section V and discuss possible improvements in Section VI. Finally, we conclude our work in Section VII.

II. RELATED WORK

A. Middleware Systems

Virtual worlds are highly-distributed systems whose internal communication may be carried by one of several approaches.

Many systems start by defining the exact wire protocol of messages to be exchanged. Typically, these systems define their own functions to create and send those messages. Second Life [1] is particularly known for having created many messages that are highly specific to the functioning of the world. Fixing the wire protocol makes these systems not only fragile in terms of changes to the application or the protocol, but also hard to change and extend in general.

Other systems use a middleware layer that defines the interface (usually in the form of an Interface Definition Language, or IDL) to be used in the application and hides the concrete network implementation. Still, most of these systems employ a 1-to-1 mapping of IDL messages to a concrete wire format and transport mechanism (e.g. TCP/IP, UDP). A large number of those systems have been developed over the years (e.g. Sun-RPC, CORBA, xmlrpc, DDS, Thrift, ProtocolBuffers, Arvo, etc).

These systems compile the interface definition into concrete, language-dependent data structures and corresponding data marshalling code to be used when sending messages or calling remote functions. This has several issues: (i) it requires the application to copy data into these middleware-specific data structures, which complicates the application logic and adds overhead for copying, or, to avoid this overhead, (ii) forces the application to use the middleware-specific data structures internally, contaminating the application with middleware-specific constructs. (iii) Because these systems do not know the data layout of the application, they are prohibiting highly efficient memory-mapped data communication mechanisms like RDMA, which can avoid the entire OS and network stack overhead, and offload almost all communication to a dedicated network hardware. The latter mechanism is particularly important when considering...
the performance of the server-to-server communication for highly scalable virtual world architectures [4].

Some middleware layers offer dynamic data interfaces, e.g. CORBA [5] and Apache Avro [6], where the internal representation of the data is created incrementally and at run time from data provided by the application. This approach has significant additional overhead due to many recursive function calls where each one typically just copies a single primitive data type or calls other such functions. Additionally, dynamic data interfaces often require developers to write more code as no custom interfaces are present and generic ones must be used instead.

B. Existing Virtual World Protocols

Virtual environments having the largest number of users are typically commercial online games based on proprietary protocols. Often, there is little to no public information available about them. They usually deliver good quality of network utilization and run well with the applications they were designed for, but they are not compatible with each other and are not designed to offer generic communication or support additional / unknown interfaces.

Second Life [1] is one of the few examples of commercial applications that have disclosed their protocol, which facilitated the development of alternative clients that can connect to the virtual world. This protocol is implemented in application logic and completely defines how the messages should be encoded on the wire, which messages are to be sent, and in which order. Many messages are very specific to the Second Life virtual world and decrease the freedom of implementation in the client. In Second Life, the Capabilities interface allows the protocol to be extended with messages defined in an XML file in a similar way as described in the aforementioned middleware. Despite this, the base protocol must still be implemented by every client.

MPEG-M, also known as the MPEG eXtensible Middleware (MXM) [7], is a suite of ISO/IEC standards [8] developed under the supervision of the Moving Picture Experts Group (MPEG). MPEG-M was designed to provide a middleware architecture for the Internet Protocol television (IPTV) technology, enabling developers to create interoperable service components running on all MPEG-M compatible devices. The domain of services provided by MPEG-M is suitable for building virtual worlds. It includes APIs for creating and processing audio and video content, as well as for user identification, location, 3D graphics, etc. To develop services, MPEG-M provides a predefined set of APIs implemented by the MPEG-M engines. Developers may only use these specific engines though, which may not be appropriate for their needs and does not allow the addition of novel virtual world features.

A much less specific protocol is used by the Sirikata platform [3]. It is based on the previously mentioned Protocol Buffer middleware and provides .proto definitions for basic messages that allow connecting to the server, for synchronizing time and object location, for discovering new objects and delivering messages to other objects. To provide the necessary flexibility, arbitrary protocol extensions can be added simply by defining new Protocol Buffer messages. There is no default mechanism for discovering whether a certain object supports an extension and the typical approach is to define some introduction message, which initiates the communication on both ends and confirms whether the other party understands this particular protocol extension or not.

III. An Open Modular Middleware

When designing a middleware for virtual worlds, we need to consider the specifics of the virtual world communication. In particular, virtual worlds typically have various message types that need to be delivered with different reliability and latency requirements. For example, a money transfer message needs to be delivered reliably, but a delay of a few seconds can be tolerated. On the other hand, an avatar position update message needs to be delivered with minimal latency, but the loss of one such message is not significant.

A. KIARA

The systems mentioned in Section II typically offer the same delivery guarantees for all messages or require implementing such guarantees manually in the application.

The KIARA middleware currently being developed and used prototypically in this paper overcomes the limitations of these systems in the following ways:

(i) it clearly separates three different concerns in middleware systems: the IDL, which describes the contract between systems and what data needs to be transmitted, is differentiated from both the API, describing how the application makes the data available and initiates communication, and from the protocols and transport mechanisms that define how the data is formatted and sent when talking to a remote system.

(ii) It allows an application to describe the layout of its relevant internal data structures from which the data to be communicated is extracted by the middleware in an optimized way when needed.

(iii) It allows the communicating systems to negotiate the exact protocol and transport mechanism to be used for communication.

(iv) It uses an embedded compiler to create optimized code at runtime for accessing the application data and formatting it as necessary for the negotiated communication mechanism. An alternative but optional interpreter allows for trading performance for increased portability and code size.

(v) It offers high flexibility in supporting new interfaces. Services can be asked for additional interfaces they support allowing for adaptive handling of complex
virtual worlds where not all capabilities of all objects in the scene need to be known to all connecting clients.

A full description of the KIARA interfaces and its API is beyond the scope of this paper, but we present the basic principles and extensions that are relevant in the context of communication in virtual environments.

KIARA uses an IDL file with .kiara extension to describe data types and services used during the communication. The current KIARA IDL is based on the Thrift syntax [9]. One major extension is the support for annotations based on the Web IDL syntax [10]. Annotations can be used to specify various properties, such as delivery guarantees, mark functions that are translated into one-way messages or specify endpoint URIs or protocols as shown in Figure 1.

Another important distinction from the other systems is that KIARA offers an API that allows developers to pass native data structures directly into the library. The code for serialization and de-serialization will be generated at runtime using a built-in compiler. Please refer to Section IV for more details.

This design allows us to perform many optimizations that were not possible with other systems. At runtime, KIARA obtains the information about the protocol support on the remote end, quality-of-service requirements and any other information that can be used to choose an optimal protocol and corresponding serialization/de-serialization code on both ends.

B. Non-KIARA-based Systems

Sometimes it is not possible to modify an existing server or client software to add KIARA support. This may happen when this software is developed by a different organization or when it is already widely deployed and difficult to upgrade. The design presented in the previous subsection allows for connecting to remote systems without KIARA support. This is achieved by generating a serialization code that transforms native data structures into the protocol supported by the target.

In Figure 2, we present an example protocol description mapping a chat service to the Protocol Buffer implementation in the Sirikata virtual world. Here, optional annotations are used to add required meta-information. Firstly, we annotate the service with ProtocolBuffer that specifies the URI to the .proto definition file where definitions of the Protocol Buffer messages can be found. Then, to map each call to the specific messages, we need to add PBRequestMessage, which contains the name of the request message, and, optionally, PBResponseMessage, which specifies the name of the response message returned from the remote end. In Figure 2, we use a function that does not need to require a response message, which is denoted with Message annotation, thus we omit PBResponseMessage.

Finally, to map function arguments to the specific fields in the request and response messages, we specify a type mapping string in the PBTypeMapping annotation. It uses the same syntax as a type mapping string used in a KIARA application. Please refer to Section IV for more details.

C. Modular Protocol

In order for virtual world applications to communicate using KIARA, one needs to define IDL interfaces that are supported by all parties. To be able to do so, one would need to define a set of services that is a superset of all services in all virtual worlds. However, as virtual world protocol research is still very active, there are various protocols that often use different synchronization concepts and interfaces. To make our system as generic as possible, we require any KIARA endpoint to support only the interface query mechanism, which allows to dynamically query which interfaces are supported and use the ones supported by both parties. While this doesn’t solve the problem of incompatible protocols, it lays the foundation on which various interfaces can be built and dynamically requested. This allows to write client or server applications compatible with multiple protocols and that are able to connect to various virtual environments.

A basic interface would entail a function listing the available interfaces, however it only makes sense to request those interfaces that are already known to the caller. Therefore, our interface query mechanism is very simple and described as a KIARA service – see Figure 3. Interfaces are identified by

Figure 2. A simple service for sending chat messages. Optional annotations are added that map this service to the Protocol Buffer protocol.
Figure 3. Dynamic interface query mechanism, which allows a virtual world application to discover which interfaces are supported by the remote end. Using this service, it is possible to write an application that supports multiple protocols and can connect to otherwise incompatible virtual world environments.

IV. IMPLEMENTATION

We present below the details of our implementation, although alternative design choices are certainly possible. Moreover, custom versions may be designed for each programming language to take advantage of their syntax and programming paradigm. As part of this work, we have designed C#, C++ and JavaScript APIs, but only present the latter for brevity.

Also, as mentioned previously, in virtual environments, the communication is bidirectional. For this purpose, our server and the client APIs are identical and may also be used in other architectures (e.g. peer-to-peer). In Figure 4, we present a short snippet illustrating the usage of the API that opens a connection, registers a local implementation for the function chat.sendMessage (previously defined in Figure 2), generates a remote function wrapper and calls it.

Part (a) shows how to create a context. It is used to store and manage shared settings for all further created connections (e.g. default error handlers). Using the context, one can open a connection as seen in part (b). The first argument to the openConnection method is a URI that points to a description of this service, and the second is a callback that is executed when the connection is ready to be used.

Later in parts (c) and (d) of Figure 4, we demonstrate how to register a local implementation for a function offered by the service and how to generate a function wrapper and then call it. Such a call returns an object that can be used to assign up to three different handlers to the function call: success, which is executed upon successful execution of the function on the remote end, exception, which is invoked when an exception is thrown while executing the remote function, and error, which flags any other error, such as when connection was interrupted or no timely response has been received. Optionally, the developer may also call wait to make a synchronous call.

Both of these use cases allow the developers to pass a type mapping string as a second argument to the methods registerFuncImplementation and generateFuncWrapper. This string allows to map fields in the native structures to the fields in the structs defined by the .kiara definition file. The type mapping string consists of a set of pairs that, for values from the IDL, specify where to find the application values in the native data. This concept is illustrated in Figure 5 showing a sample IDL file that defines a structure KiaraStruct and a simple service myService with one function myFunc taking the structure as one of two arguments. This service is then used in a JavaScript application that relies on a different layout of the data in the structure and a different order in which parameters are passed to the function. Instead of rewriting the IDL file, which may already be used in other applications, or changing the application itself, which may require substantial work, we map the data structures to one another using a type mapping string passed to the generateFuncWrapper method. In particular, we map two string values stored as an array in the native field field1 to fieldA and fieldB in the KiaraStruct,
result are an array of parameters and a return type of the IDL function, Args and Result are an array of parameters and a return type of the native function or function wrapper and thus corresponds to the native types. When the developer maps an entire structure to another, KIARA will try to find fields with corresponding names and map them automatically. This is especially convenient if the .kiara interface was designed after the native type.

V. RESULTS

In order to evaluate our approach, we have modified the OpenSIM virtual world server [11] and implemented two prototype clients as HTML5 browser applications: a virtual world chat and a basic client viewer.

To illustrate the ability to dynamically check supported interfaces, we have split the default Linden Lab protocol into a set of interfaces, which are dynamically queried from the server and the client. This allows to provide a different set of features on various clients or add new features to the server while maintaining the compatibility between them. For example, the chat application supports the login, connect and chat interfaces, while the viewer client offers the login, connect and objectSync interfaces. The login and connect are essential for each client and needed to authenticate into the virtual world and to exchange the initial world and user data upon startup. The objectSync interface allows the server to deliver a 3D representation of the virtual world objects in the vicinity of the user’s avatar, and the client to continuously update it’s avatar position on the server. Finally, the chat interface adds support for exchanging text messages in a virtual world chat. The client queries the required interfaces from the server using the interface query mechanism, while the server uses the same mechanism to query supported interfaces from the client and to determine which set of its features and services should be enabled.

When implementing the viewer client application (see Figure 6) we used the Open Modular Architecture for Virtual Environments [12] due to its flexibility and good integration with other Web technologies. This architecture uses the XML3D technology [13] for 3D rendering. Instead of converting 3D content from the format used in the Linden Lab protocol, for the prototype we have added a new interface that has allowed us to deliver mesh data in the XML3D format directly. However, it is possible to convert the 3D format as well and we are planning to do this as one of our next steps. Thus, we have demonstrated the extensibility of the protocol and ability to support novel solutions.

Moreover, in the chat client, we have added support for a second virtual world platform – Sirikata, which, as mentioned before, uses the Protocol Buffers for communication, with a custom layer of reliability. The most efficient way to add such support would be to implement KIARA on the Sirikata server, which would help both systems to choose the most optimal protocol. However, to demonstrate that our technology can be used with an existing unmodified system, we have implemented this custom protocol in KIARA on the client, which has allowed us to create a chat application capable of connecting to both the Sirikata and OpenSIM virtual worlds as seen in Figure 7.

Finally, there are two reasons why our method is simple to program. As shown in Section IV, one can pass native data structures into the functions and the serialization code is generated automatically. In particular, while implementing KIARA services on the OpenSIM server, we were able to pass native data structures to the generated function wrappers directly and the KIARA runtime automatically serialized them to the wire protocol. Moreover, the wire protocol is automatically and optimally chosen at runtime,
and developers do not need to manually implement the selection logic.

VI. DISCUSSION AND FUTURE WORK

Although the current implementation of our prototype library demonstrates the overall potential of our approach, a number of further developments are necessary in order to make the system applicable at a broader scale.

To make it easier for the researchers to work on new solutions, one would need to port our prototype library to a larger number of programming languages. Additionally, some of the existing implementations are incomplete and only support one fixed JSON-based protocol, which may be suboptimal for virtual worlds. Solutions to this limitation are already being developed in the C++ version of KIARA, whose early version of the compiler can optimize the underlying wire protocol at runtime, taking into account the current network configuration, security and quality-of-service requirements. When this research is completed and ported to other implementations, we believe it will make our open modular middleware even faster and more efficient.

Finally, we intend to design a listening API for KIARA. Our prototype library reused the existing functionality provided by the application we’ve extended. However, we believe that providing such an interface will facilitate the development of new applications based on our API by the research community at a larger scale.
VII. Conclusion

In this paper, we have presented a novel communication solution for virtual environments – an open modular middleware. We have shown that it allows developers to dynamically request supported features, is extensible to support new technologies, can be used with existing unmodified systems and is simple to program. Our solution is a good foundation on which new interfaces can be built with great potential to connect previously incompatible virtual world platforms.

During the course of this work, we came to realize the extent to which numerous common concepts are actually shared by the protocols of different virtual worlds. This high degree of homogeneity can be seen as a sign that the design of universal interfaces is indeed feasible, and we regard the work presented in this paper as a step forward towards reaching this ultimate goal.

Acknowledgements

We would like to thanks the creators and contributors of the OurBricks [14] and The Free 3D Models [15] websites, from which we have downloaded 3D models for our basic client viewer application. We also thank the contributors of the OpenSIM project, which we used to create our prototype server application.

References


