Inter-Process Communication for Digital-Human Model Inter-Connectivity

Andrew Taylor and Tim Marler*

116 Engineering Research facility, Center for Computer Aided Design, University of Iowa, Iowa City, Iowa 52242

Abstracts: Digital human models (DHMs) provide tools for simulating human performance and capabilities, which allow users to evaluate any product or process virtually. As applications for DHMs expand substantially, so does the necessity for more complex and more accurate models. This in turn requires coupling of sometimes co-located sub-models for simulating various aspects of the human. In addition, DHMs are used in conjunction with other types of (non-human) models in order to evaluate products and processes, and this also necessitate integrated systems for seamless concurrent engineering. This level of multi-scale modeling and human systems integration requires a new inter-process communication (IPC) system specifically for DHMs. Thus, this paper provides a new architecture for peer-to-peer IPC. The proposed system uses a TCP/IP protocol for necessary security and for ensuring all data packets are received in order. The system has been designed to facilitate an uninterrupted workflow, which is critical for applications that involve concurrent team-oriented analysis and design. In addition, this new system supports multiple connection mediums and facilitates variations in the number of users. The common issues of blocking and potential lost data have been addressed. Test cases demonstrate superior computational speed.

Keywords: Inter-process communication, Interconnectivity, Digital human modeling.

INTRODUCTION

Almost all products and processes involve some level of human interaction, and most products and processes are modeled virtually (on a computer) during the development cycle. This points towards the necessity for digital human models (DHMs) that can virtually predict human motion and performance. In addition, such models can simulate physiology, effects of changes in anthropometry, potential injuries, etc. However, as one attempts to model more complex systems, depending on and growing a single model becomes less tractable. As applications and necessary capabilities expand, the demands on DHMs can no longer be met with individual models.

Digital human modeling provides a framework to assess the effect of task performance as well as various physical impairments, including weakness, fatigue, and limited range of motion. Consequently, higher fidelity and more accurate human models are becoming critical for meaningful simulations used for injury prevention, for product and process design, and for the study of human function. However, increasing the fidelity and accuracy of a model of something as complex as the human body requires a multi-disciplinary, multi-scale effort. Often a system level model is needed for simulating task completion, and somewhat separate highly accurate computational mathematical models may be needed for simulating subsystems (*i.e.* knee joint, physiological systems, etc.). Although the field of digital human modeling has gained significant momentum and the medical field has grown in terms of modeling and simulation success, there are deficiencies with currently available human modeling tools. Most of these human models, which are geared towards simulating whole-body activities, simply do not have the fidelity or the broad range of capabilities necessary to address issues associated with various types of injuries. In general, especially considering the complexity of the human body, moving towards a comprehensive human model requires many interconnected models.

For example, as detailed by Marler and Sultan [11] and Sultan and Marler [12] and illustrated in Figure **1**, predicting orthopedic injuries accurately requires integration of models for simulating gross human motion, motion dynamics, muscle response, orthopedic stress and strain, and propensity for injury, as well as neural networks for creating a meta-model of the finite element model, which can be relatively slow.

In addition to linking various models in order to form a multi-scale model with a collective increase in fidelity, there is also a growing need for interconnected models that foster concurrent design and analysis. Marler *et al* [10] present an example of this kind of system integration for survivability analysis. Capabilities are linked/integrated for human mobility assessment, ballistic hit detection, motion of internal viscera, and

*Address correspondence to this author at the 116 Engineering Research facility, Center for Computer Aided Design, University of Iowa, Iowa City, Iowa 52242;
E-mail: tmarler@engineering.uiowa.edu

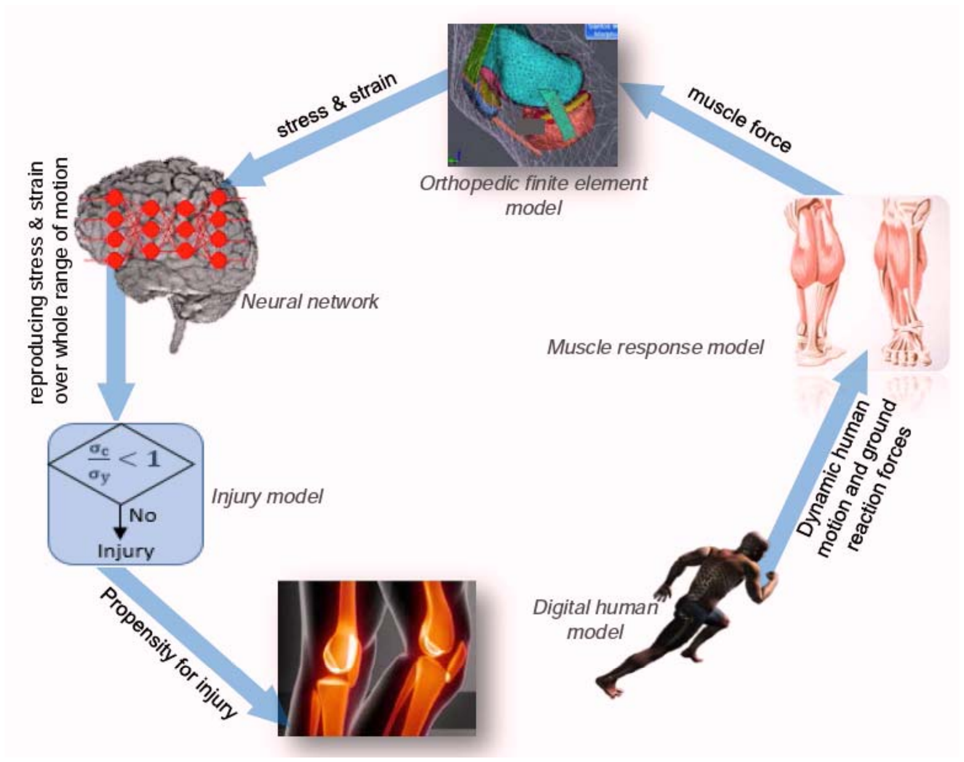
stress & strain
muscle force
Orthopedic finite element model
reproducing stress & strain over whole range of motion
Neural network
Muscle response model
$\frac{\sigma_c}{\sigma_y} < 1$
No
Injury
Injury model
Dynamic human motion and ground reaction forces
Propensity for injury
Digital human model

Figure 1: Example of Interconnected Models for Digital Humans.

body-armor optimization. Accordingly, there are substantial opportunities for advancements with multiscale computational predictive methods. However, such advancements require an inter process communication (IPC) system specifically for digital human modeling.

The proposed work capitalizes on, and adds to a foundation of virtual human modeling capabilities, housed within a DHM called Santos [1]. Santos is a highly realistic, physiologically mechanistic, biomechanical computer-based human that predicts, among other things, static posture, dynamic motion, joint strength, and development of fatigue. Such capabilities can be used to predict and assess human function, providing task performance measures and ergonomic analysis. Thus, in a virtual world, Santos can help design and analyze various products and processes. Although functionally mature and able to serve as a virtual design and analysis aid, Santos continues to grow as a multiscale human model, with efforts to incorporate clothing models, link with blast and ballistic survivability models, and increase computational efficiency of dynamic motion prediction.

Specifically, we propose a general IPC system that further enables this expansion. Essentially, this will allow multiple software systems to exchange data easily and immediately, rather than requiring one to e-mail text files for instance. Indirectly, IPC will allow any number of users, located anywhere on the globe, to work on the same project concurrently. We contend that although no work has been completed with seamlessly integrating DHM sub-models, a well-designed IPC library can in fact facilitate a multi-scale, multi-model DHM.

IPC is a catch-all term used to describe data exchange and communication between different programs. There is no single method for implementing IPC, which is largely dependent on the operating system (OS) under which the host applications execute. The most exciting development with respect to the IPC is the potential use of Windows sockets for connecting multiple users, potentially in different locations, simultaneously. In addition to allowing immediate exchange of data and analysis results, this capability will allow basic operation of applications from different external applications.

In this paper the term *IPC Library* refers to the independent volume of code written to support our IPC solution as described. This library can be included into third party applications, and is also included in our own software. It contains all the functionality described in this document to facilitate Inter Process Communications. It should be noted that in most cases a “Network” or “Communications” library is a general purpose code that allows different applications to communicate over networks, in a common or generic way. The term IPC is specifically used to denote a less general method, where communications are targeted between specific types of application that share specialized and possibly proprietary data. Inter Process also describes multiple applications communicating rather than specifically using networking. Two applications could share data on the same PC for example without using traditional networking, instead using an alternative method such as file sharing, or sharing features of the underlying operating system.

The idea of IPC falls under the broader topic of networking. In 1970 the first successful distributed, wide-area, hardware agnostic networking system was demonstrated [2]. The system, which was called ARPANET and was funded by the Defense Advanced Research Projects Agency (DARPA), linked together four nodes: the University of Santa Barbara, UCLA, SRI International, and the University of Arizona. ARPANET connected systems with different software running on different hardware platforms, and facilitated data exchange [3]. To achieve this goal a set of detailed networking protocols were developed that each node subscribed to. By 1972 the ARPANET “packet” protocols were used as the base technology to create what would become today’s Internet. The original ARPANET protocols were expanded [4], to include error checking, and the basic TCP/IP (Transmission Control Protocol/Internet Protocol) architecture was developed [5].

The common term for this kind of technology is the Internet Protocol Suit, which is a computer networking model consisting of a set of layered protocols. TCP/IP describes how data is packaged, addressed, transmitted, routed, and received at the destination. The transport protocol layer is the most important for the IPC library. TCP and UDP (User Datagram Protocol) are arguably the most commonly used transmission protocols. TCP provides reliable, ordered, error-corrected delivery of data over an IP system of servers. It includes rules to resend packets that get lost, automatic resends for corrupted packets, and ensures that packets are received in the order they are sent. However, TCP is less efficient than some other protocols. Alternatively, UDP is a faster protocol, but is less reliable. It performs no error checking beyond data checksums. It has no handshaking dialog, does not guarantee delivery, does not guarantee ordering, and has no mechanism for resending lost packets.

TECHNICAL BACKGROUND

Common Challenges

Even given the most appropriate protocol, networking often faces two common challenges. With business and internet communication models, the current paradigm involves large amounts of data securely sent and received, lossless data delivery (no data lost along the way), and reliability. Efficiency and speed are secondary concerns. These models also follow a particular software design whereby errors are blocking problems; the software stops requiring user input. Using the example of a web browser, this difficulty is an inherent feature of the design. When a page on a server is requested, the browser waits, continually listening for a reply from the server. If it receives a reply, the connection is made, the page loads, and the browser continues running. If there is an error such as the server not responding, the browser times out and an error page is displayed. The software comes to a halt awaiting user input. This is also demonstrated in other enterprise software, where the application halts when a data transfer operation fails. Sometimes blocking is acceptable when the application depends on a network operation to be completed. However, in many situations, including common uses of DHMs, blocking can be overly restrictive. DHMs depend on a continuous workflow including real-time animation, analysis, and frequent non-blocking user input. In such an environment, IPC has to work in a non-blocking way, but data transmission must remain reliable. An example of this non-blocking behavior exists in networked video games. Networking in video games however, began not with the internet, but with local network hardware.

PC hardware is the second factor in developing high speed networking, especially inexpensive, mass-producible hardware. The most successful networking hardware is called Ethernet [6]. Work began on Ethernet technology as TCP/IP was being finalized in 1976. By the early 1980’s Ethernet was a fully viable hardware implementation for local communication between individual computers. In 1995 Microsoft released Windows 95 on IBM compatible PCs which supported Ethernet peer-to-peer LAN, (Local Area Network) communications. This coincided with (or spurred) a flood of cheaply produced, reliable Ethernet cards. A year later, a game called Doom began the networking gaming revolution still popular today [7]. There were notable examples before Doom, but Doom was the first mass appeal LAN game. The main problem in games and continuous, real-time applications like Santos, is that communications have to be fast and non-blocking. If networked games slow down, or worst still, stop and wait for data to be received, the game becomes unplayable. The communications also need to be as continuous as possible because the software runs at a high frame rate. Thus, UDP in games is a better choice than TCP, because it is more efficient. However, without much delivery and ordering protection, game networking is forced to accept unreliable delivery. This is one feature we do not want to deal with in IPC, at least not on a packet level. This eventually became known as a lossy (data lost along the way) non-blocking system. In our IPC design the non-blocking nature of Doom works well, but not lossy UDP transfer. Our design instead uses TCP/IP which, while slower, affords us more reliability and security, and is lossless.

Selecting a Model

The next step in the design process is to select a networking model. There are two major models for communications networks: peer-to-peer and server networks. The best example of a server network is essentially the internet. Internet servers are distributed geographically connected together in a haphazard pattern called a “web”. Servers can have multiple connections to each other (not always based on proximity). When a user connects to a web site, the data is routed through the server network and then back to the user. This is called a distributed design. Multiple connections are maintained through servers providing many alternative routes. If a server stops functioning, communication is rerouted in a different way to reach the target. Reliability is a function of how many servers a network has, and how many connections there are between servers. The primary disadvantage of a server based system, is that if the first server the user connects to goes down, (usually the user’s Internet Service Provider) the user cannot connect to the network.

With peer-to-peer networking, computers are connected in a haphazard network maintaining connections with other peers. However, every peer acts as both a host server and a simple user. This means that as long as a computer is connected to at least one peer, it can always access the network. Under the internet model, it would be similar to a customer having multiple ISPs for the same price. If one ISP goes down the customer can still connect through one of the others. The disadvantage is that the peers in this network are not efficient, dedicated servers; they are simply desktop computers running the same networking application. This limits the type of content and the amount of data the network can carry. Running a large file service through a peer-to-peer network for example is inefficient. However, using a peer-to-peer model on a DHM network, which is a relatively small scale application, frees us from unnecessary overhead in terms of complex server software and difficult network management.

Current Protocols

There are many currently available solutions (protocols for packets) that handle different kinds of communication problems. However, none fill our specific design requirements. Most modern designs have similar features. One typical solution is ATP (A Transport Protocol) [8]. This assumes that large amounts of data will be sent and received across the communication software, and that the way to optimize transmission speed is to optimize data congestion. When sending large volumes of data, a communications system can quickly become bogged down (both in hardware and software), as buffers fill up and the system waits for older data to be cleared out. This protocol optimizes transmission and receivership of data to minimize data collision and congestion.

Another common approach is to use the UDP protocol, but as previously mentioned, UDP has some reliability issues. The challenge is to retain the reliability of TCP but keep communications speed fast. An example of this is the UDT (UDP Data Transport) protocol [9]. This model contains many of the usual features in modern network design; namely features to speed up communications. The focus again is on transmitting large volumes of streaming data, which requires memory efficiency, congestion control, rate control, and future traffic prediction. The library appears to be faster than using TCP alone, which is a noted achievement, but it still contains some inherent disadvantages of UDP. The authors note that they have not accounted for all of the disadvantages of UDP and merely assert that they have not seen any problems in those areas. Nonetheless, this is a critical design flaw. Santos is commercial and applied experimental platform. It is important that under real world stress, the proposed communications hold up. Efficiency is a concern, but not enough to risk using UDP.

Additional methods tend to be variations of the same themes: how to send large amounts of streaming data while avoiding traffic congestion, packet loss, or memory loss on the local PC.

METHOD

We considered this problem from a new perspective. We approach system integration and communication from a DHM perspective. What do we need to communicate over a network? How often does this communication occur? How much data is involved? How dependent is the software on absolutely receiving that data? Starting with these considerations, we designed the IPC library not from the communications aspect of sending large volumes of data as quickly as possible, but from the software requirements of a DHM application like Santos.

This distinction is perhaps the most important feature of our design. Designing a generic communications engine that functions in all cases is not necessarily the goal. It is not possible to craft a one-size-fits-all solution. Given this overarching approach, the proposed design has the following features:

A Peer-to-Peer Model: Lightweight, easier to manage, lighter in infrastructure and code, and more efficient for smaller amounts of data than more complex server-to-peer based systems.

Use of the TCP/IP Protocol: Although there are faster protocols available such as UDP, TCP/IP provides necessary security including data corruption protection, automatic resends of missed or corrupted data packets, and ensures that all packets are received in order (something not guaranteed with other protocols).

A Non-Blocking Model: The IPC library and the application using the library must not disturb or interrupt a continuous operation. In this respect, we look to the lessons learned in the games industry regarding packing data, how processes are constructed in the code to avoid delays, and splitting up and prioritizing data to eliminate blocking situations.

Graceful Error Failure: In tandem with non-blocking operation, communication operations that fail must also be non-blocking. The software must handle failures in a manner that does not require user input, or halt execution of the program.

Dependent or "Staged" Operations: We acknowledge that sometimes blocking operations are necessary in multi-stage, peer dependent operations. This is especially prevalent when performing any kind of shared static analysis with a DHM. An analysis might involve processing some state of an avatar, sending it to a peer for further processing, and waiting for those results to come back before more work can be done. Most DHMs have some processes such as these that can take hours to process. There must therefore be a method to support a blocking form of staged execution, but not to the detriment of continuous operation.

Support Multiple Connection Mediums: As previously mentioned, the goal of IPC is to connect multiple applications together, not necessarily to network them using any specific method. The library should be designed in such a way that different mediums that connect applications together can be added, removed, and revised over time. To support this goal we engineered the library supporting two very different communication methods: Internet networking, and local file I/O.

Ease of Implementation in DHM Software: DHM software comes with its own set of difficulties and specialized programming disciplines. Many programmers in this field may not be comfortable with third party networking libraries, or simply do not have time to research this topic. The library therefore must be easy to implement by a programmer, employ a simple interface, and provide a comprehensive SDK (Software Development Kit). The SDK must include example source code and documentation detailing the implementation and functionality of the library.

Architecture Overview: In this section the method of implementation is discussed. This includes the architectural fundamentals of the software, and operating system features that will be leveraged.

Two layers of software are required in order do achieve our plan: a low level layer that controls the specifics of connecting, sending, and receiving data, and a higher level that works in cooperation with the Santos application (or third party application) to provide synchronous and asynchronous operation. The lower level, (which has been referred to as the IPC Library thus far), can in effect work largely independently acting as a service to the larger application. It does not have to contain any formal specifications of the type data transmitted. The higher level layer however, must have some knowledge of the type of data being transmitted/received, or at least understand how the process using the data will be handling the response.

Basic System and File Handling: The IPC library will provide communications support to an application, allowing it to send and request data from other applications. In this respect the code is not dependent on the DHM architecture and will be built as a re-usable and generic code. This will allow a third party application to include the library as simply as possible. The library only supports basic communica-tion, designed to hide from the user the complexities of sending and receiving data. What the client program does with the library, and the type of data sent and received is entirely up to the developer.

The library will also present a common, simple, generic interface that supports communication using any number of methods internally. This allows us to expand the library in the future to include newer or faster communication methods without breaking the programs already using the library.

Windows Socket Layer and Integration: The main target for communication within the library is Windows Sockets. This is a standard library built into the Windows operating system, which allows data to be sent/received through the Internet, or across an internal LAN based network. Windows Sockets use standard Internet (IP) protocols to communicate just like a web-browser. These methods are highly reliable, well tested, and have been available since 1982. Windows Sockets applications however are often difficult to architect and require significant testing under various conditions, and this could hinder development for all clients waiting to use the library. To mitigate this risk a file handling strategy is implemented first.

File handling uses basic IO files on a local or network disk to share data between clients of the library. This method of communication is slow, but much easier, quicker, and less bug-prone to implement than full Sockets development. By implementing this method first, even though it is slower than Windows Sockets, clients receive an early version of the library that supports the major features for communications.

Low Level Software Architecture

In this and the proceeding section, the software architecture is discussed. The software is split into two sections. The lower level is agnostic regarding higher purpose. It simply supplies the base system for physical communication. The higher level system specifically provides the DHM networking solution and is dependent on the library.

The lower level software performs the physical task of connecting, disconnecting, and sending/receiving data. This layer is developed as a standalone library of code. In this case standalone means that the codedoes not rely on any of the Santos SDK (Software Development Kit) or API (Advanced Programming Interface). Also, to make the library as flexible as possible, it iswritten in the widest available language in use today, C++. As Santos is written under the Microsoft Windows operating system, the library is written in Visual Studio (2010), using the standard Windows libraries (no MFC). The TCP/IP communication is facilitated through the standard WinSOCK implementation (version 2.1 and above). Although the library is .NET compliant, it is not .NET dependent and is compiled into a standard static .LIB, as well as a windows dynamic .DLL file. It does not contain any managed C++.

Santos uses a mixture of C# and C++, the major portion being C# under .NET 4.0 (at the time of writing). Thus, a C# file is included in the SDK for the library, which provides a marshalled C style functional interface. In this way the library can be used in both C++ and C# projects. Due to marshalling and copying of unmanaged memory to managed memory, using the library in C# is less memory efficient. Execution time is also effected, as there is a transition penalty across the managed/unmanaged execution boundary.

There are three main objects that comprise the system. These are the Gateway, IConnection and IDriver classes. These classes are described in this section to illustrate the architecture of the IPC library, and how the problem of future expansion, and multiple communication mediums are solved.

Perhaps the most important class is the I Connection class (Figure 2). This base semi-abstract class describes a named connection associated with an abstract parameters class. It supports a standard interface to manage a generic communication medium. The generic interface provides methods to initialize (or open) the system, connect to peers, send and/or receive data, disconnect from peers at some future time, and finally close the system and release any resources.

Both incoming and outgoing connection types are derived from this class. Connections have a threading mechanism for any potential blocking operations. The deriving class implements the virtual Thread Update method to take advantage of this feature. A system-wide MUTEX controls thread prioritization, thread handles, and calls to individual Thread Proc methods when connection classes are initialized.

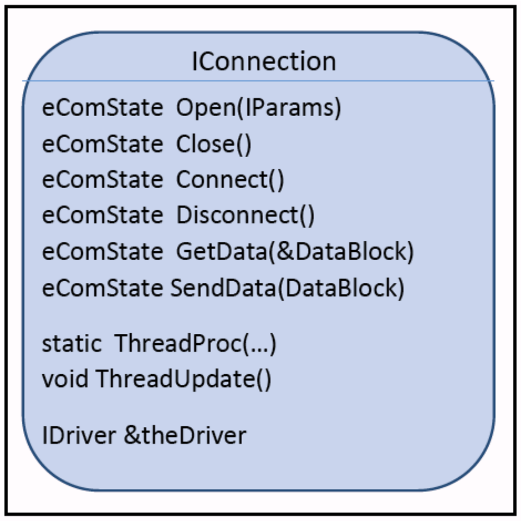

Figure 2: The major virtual and static methods in the IConnection class.

Each connection also contains one or more IDriver classes (Figure 3). The driver class encapsulates the low-level (possibly hardware level) specifics of a particular communication method. This gives maximum configuration flexibility in the C++ code. A reusable drive that communicates in a specific medium, can be coupled with a connection class that sends/receives data in a specialized way. The connection could support a specialized protocol, buffering mechanism, compression algorithm, data check summing, or encryption/security encoding. However, none of these mechanisms change the basic communications code in the driver class.

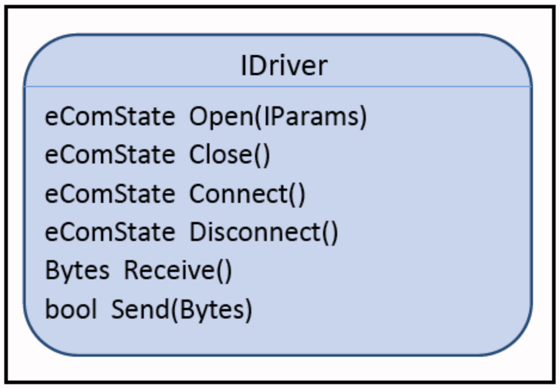

Figure 3: Virtual methods in the IDriver class.

The third major class in the library is a singleton called the Gateway class (Figure 4). Any C++ code using the library, as well as the C# interface use this class for access to all aspects of the system.

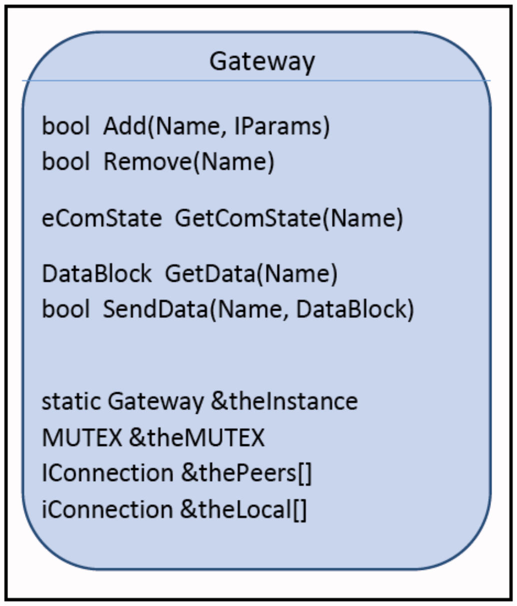

Figure 4: The singleton Gateway class.

Connection and driver classes use a small abstract data class called IParams. This class provides only one virtual method by default, to return the Local flag. A local connection is the current PCs connection. The IParams class is different for every type of connection. It contains the specific parameters required to start a connection of a required type. For a TCP/IP connection for example, a derived IParams class will include an IP address and port number. The parameters contained in IParams can be pertinent to the gateway, the connection, and the driver class. As IParams is non-instanced, any change to one of its members, changes that value for all referencing classes.

The gateway operates with named connections. Once a connection has been added to the gateway with its specific parameters, it is referenced by name from that point on wards. This makes operation easier for the third party programmer, and it makes C# interfacing possible as unmanaged pointers cannot be shared with managed C#.

As each peer is added to the gateway, an IConnection classe is created bound to the correct IDriver class according to the IParams used. In this way peers can be added that communicate through multiple mediums (as supported by the library).

The local machine must also act as a server listening for incoming connections and performing handshaking and initialization operations for each of the registered peers. The local server operations also include monitoring each connection and making sure peers are still connected, have not timed out,and have not reported an error. These operations are also addressed with the same generic architecture. Server “listener” connections are written using the IConnection class. Each communication method employs an IDriver to perform the specific listening and maintenance operations. As peers are added, the Gateway class ensures that there is at least one server class for each connection type used by peers. These connections are invisible to the caller and are managed internally. If all peers use the same communications method (TCP/IP for example), then only two local connection classes are required: a peer connection to send/receive data, and a server connection to listen and error correct TCP/IP connections.

The architecture has a unique feature: peers can connect through different communication mediums, and as long as each peer can maintain a server connection with a minimum of one other peer across another medium, network integrity can be maintained. This is true because the library supports two communication methods, TCP/IP and File IO.

High Level Software Architecture

The higher level system resides in the third party application, or the Santos software. It uses the lower level library to perform all communication operations. The higher level software processes incoming and outgoing data with full relevance to the DHM software concerned. This layer provides non-blocking and “staged” blocking execution features. It should also be noted that the higher level requires some form of GUI support. The GUI allows peers to be added/removed and to enter peer parameters (names, IP addresses, etc.). In Santos, there is also a chat GUI using the built in chat system. Santos also provides a third party plugin system with support for various types of C# plugins. For the IPC SDK, a new IPC plugin has been released which supports an interface to the high level system.

The system updates itself in a threading mechanism at 12.5Hz (eight times a second). During this update period the low level library is checked for waiting incoming data. Outgoing data sent through the system is buffered and waits for the next update period. Both incoming and outgoing data packets are processed through a sub-system called the command executer.The executer system also handles some lower priority features once per second, such as handling chat packets. These are text packets that are relayed through a simple GUI window in Santos that looks like any contemporary chat program.This feature allows peers to chat to each other across the communication medium.

Each packet sent or received has a 32-bit command code. This is used by the system to look up a delegate method in the executer sub-system. Each command actually has two delegates, one for incoming packets, and one for outgoing packets. Each command is registered at start-up with a text name, a command code, and an incoming delegate method and outgoing delegate method. Commands can be added dynamically to the system, ideally from a third party plugin that adds support for a third party application that uses specific codes. The programmer of the supplied delegates writes the code to pack and unpack the data and any processing required thereafter in the two delegates. The system then publically provides a set of overloaded Send methods. The most common usage of the Send method is with parameters for the name of the peer to receive the packet, the command code, and any parameters, (for example an array), cast as a C# object. The send delegate decodes the parameters in order to prepare the data.

Incoming packets are processed in much the same way as outgoing packets, although no programmer input is required. The system update automatically puts incoming packets through the command executer. The read delegate is called for the packet’s command number, and data is read and processed by the programming in the delegate.

A third mechanism is employed to trigger a command in a peer. This is called a request packet. Request packets literally request a peer to run a command and send back the results. This type of packet has a header and may contain parameters in the data section. The system provides a Send Request method that works like the Send method. The parameters however are packed into the data portion of the packet, and a communication flag is set to mark the packet as a request packet. When request packets are received, parameters in the data section are decoded and sent through the same interface as if the Send method had been called directly. This automatically triggers the system to reply with the correct data, without any other work on the programmer’s behalf.

This process does not wait for replies, or send data and wait for conformation. The exchange of data takes place automatically and meets several of our design goals; asynchronous communication, non-blocking execution, and graceful failure handling.

Blocking "staged" or dependent operations are also possible. Each packet is numbered with a GUID (not to be confused with a Windows GUID). A unique number on the local PC calculated by 32-bit CRC and guaranteed to be unique; the GUID remains unchanged for the lifetime of the packet. The IPC system allows an event delegate to be registered with a packet when it is sent. When an incoming packet is found with that registered GUID the associated event will be triggered. A time-out can also be added at send time; when the time-out occurs the same delegate will be called with an error status. After the event, the system automatically unregisters the GUID and delegate.

Packet Protocol

All communication methods supported by the IPC library send and receive data in variable size proprietary packets. Each packet is prefixed with a small header (Figure 5) used internally by the high and low level systems.

The peer name field for outgoing packets is set to the name of the peer that should receive the data. For incoming packets the peer name will be the local connection name.

The communication flags specify important status about the packet, such as error status, acknowledgement, command request and client is busy.

The GUID is generated by 32-bit CRC and exists for the packets lifetime, no matter how many peers handle the packet.

RESULTS AND DISCUSSION

In order to provide objective results for the proposed system, a testing scenario was designed that measures the operating time specifically for the IPC Library (the low level component). Timing of the high level component is less important, as the high level system design is variable. One implementation for a high level design has been presented here for DHM software; however, the low level library is agnostic in terms of higher level purpose. At the end of this section results are presented for the additional time required to cross the execution boundary from C++ to C#, as this is significant and does not depend on how the library is implemented.

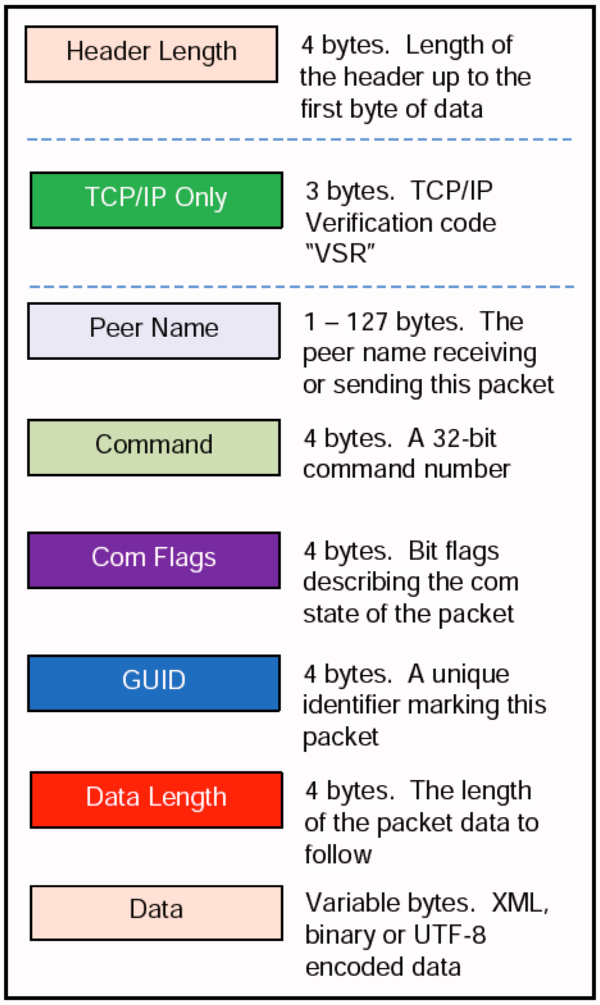
Header Length
4 bytes. Length of the header up to the first byte of data
TCP/IP Only
3 bytes. TCP/IP Verification code "VSR"
Peer Name
1 – 127 bytes. The peer name receiving or sending this packet
Command
4 bytes. A 32-bit command number
Com Flags
4 bytes. Bit flags describing the com state of the packet
GUID
4 bytes. A unique identifier marking this packet
Data Length
4 bytes. The length of the packet data to follow
Data
Variable bytes. XML, binary or UTF-8 encoded data

Figure 5: The IPC packet protocol.

The scenario used is called a "ping pong" test. A packet with a specified amount of random data is sent through the library to a specified peer, time-stamped using the Windows high resolution timer. When the packet is sent back from the peer, the timer is used again with the packet's time-stamp to calculate the total "round-trip" time. The time includes the transmission time through the medium to the destination peer and back, any operating system overhead to send and receive, and the full execution overhead for the IPC library itself, from a packets first inception into the library and back out.

The operating systems used for testing are Windows 7.1 64-bit. The testing software and the IPC Library are complied with Visual Studio 2010 and are 32-bit. All machines used in testing are Intel® i7-2600 at 3.40GHz with 8 GB of ram. Initial results are shown in Figures 6 through 7.

The results for file I/O test 1 (Figure 6) are as expected. This test uses the local PC's hard-drive for the shared files needed for communication. The overhead here is the operating system itself, opening, closing, reading, and writing to physical files. The times are uniform and unremarkable. These times would be effected if other running programs were performing disk operations, which again is as expected.

The results for file I/O test 2 (Figure 7), however, are somewhat unexpected. This test uses a networked Intranet drive for shared files. This setup requires the usual overhead for the operating system to handle files, but also requires network communication through Ethernet, as well as file handling again through a security conscious server that houses the hard drive. This test was expected to yield longer transmission times, and although this setup is slower, the change is not significant (on average < 0.03 seconds). The discrepancy was found to be Windows 7.1 networking optimization. Windows 7 now offloads much of the usual work of file I/O directly to the server. That means an entire stage of processing is missing from the local machine. The additional time therefore is the transmission time of the data over Gigabit Ethernet.

Also surprising is the TCP/IP test (Figure 8). On cursory examination times appeared faster than File I/O, which was expected. In fact most times were half that of either file I/O tests. However, periodic spikes appeared randomly throughout the testing period. After adjusting our test results for external network traffic overhead (incoming and outgoing) on the local machine, the results then look largely the same. The spikes however, do correspond highly with the local network load as reported by load tools run on a secondary PC during the testing period. Our tests were designed to remove local network congestion. We also corrected for local traffic not originating from out testing

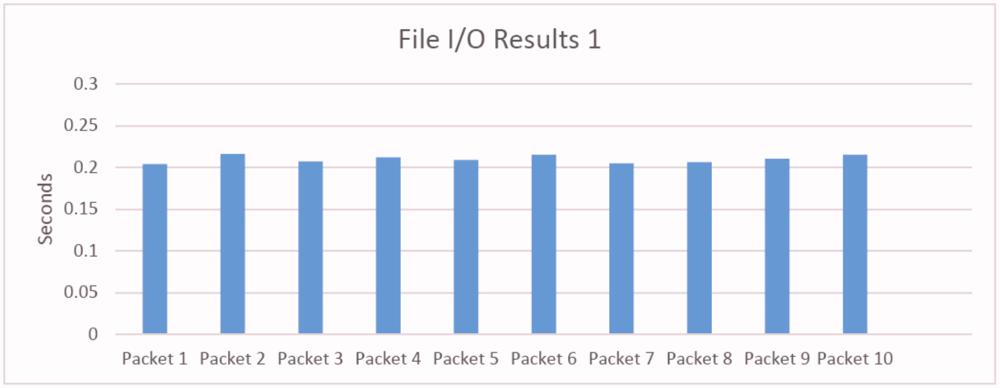

Figure 6: File I/O times over 10 packets, both peers on same PC, average over 100 packets = 0.2067 seconds, packet size = 32 KB, local hard drive used as shared path for communication.

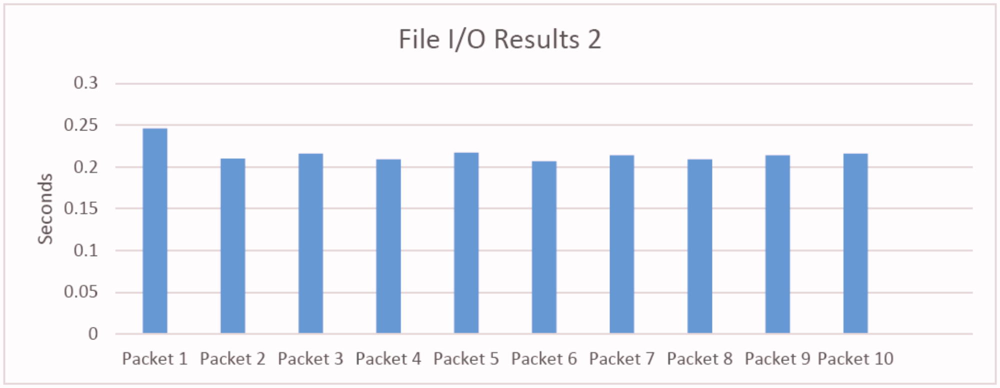

Figure 7: File I/O times over 10 packets, both peers on same PC, average over 100 packets = 0.2201 seconds, packet size = 32 KB, networked drive over intranet used as shared path for communication.

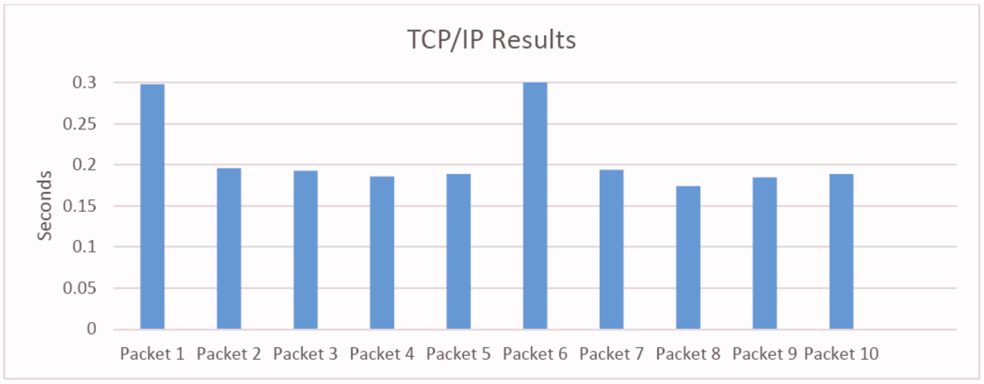

Figure 8: TCP/IP times over 10 packets, peers on different PCs over local Intranet, average over 100 packets = 0.2234 seconds, packet size = 32 KB, transmitted by standard WinSock TCP/IP.

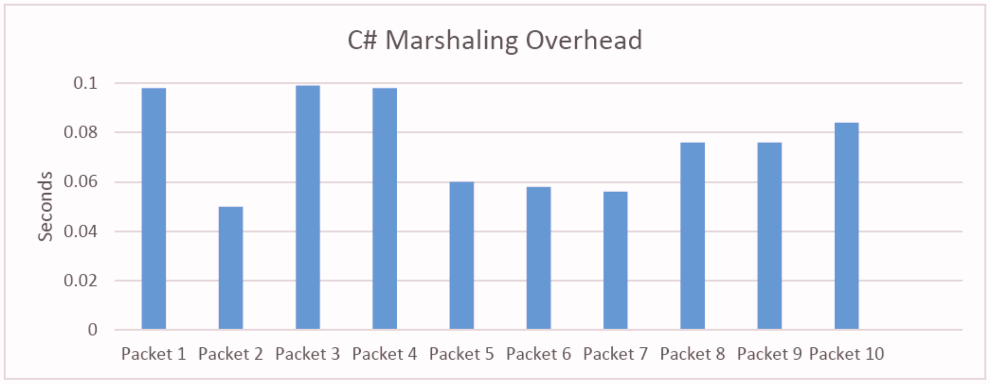

Figure 9: C# marshaling overhead for 10 packets.

software. However, we did not account for general network load beyond our control. Once accounted for, the spikes disappeared, and the average time over 100 iterations fell to 0.1792 seconds. This represents a 13% speed increase over the fastest file I/O tests.

As previously mentioned, there is a penalty for using the library in a C# program. Any data generated in unmanaged memory, such as that used by C++, must be copied into managed memory for C# through a process called marshaling. Also marshaled are parameters and structures, individual structure members, and any native pointers returned from C++. This process is memory inefficient and takes time. This final test is performed at the point a ping pong message is sent through C# to the C++ library, plus the time for the return message to be marshaled back from C++. The average time for 100 iterations is 0.086 seconds. The results are uneven, which reflects the automatic memory management in C#. Sometimes when large blocks of memory are marshaled, the C# garbage collector must make space and use some time to compress the garbage heap. Averaged over all communication methods, the penalty for using the library with a C# program is approximately 37%. This is a lower overhead than typically seen in C# applications using a significant portion of unmanaged code.

To attain this low latency, the C# interface has been highly optimized. Wherever possible, automatic marshaling is forgone. Manually returning structs containing IntPtr types is more desirable. The IntPtr types are then used as real pointers in unsafe blocks to resolve marshaling manually. No built in marshaling methods are used such as Marshal.Copy… or Marshal.Read…, or library classes such as BitConverter or Buffer. Manually using unsafe pointer logic is faster if blocks are aligned on 32-bit boundaries and are sized in multiples of 32-bits, regardless of their actual content size in bytes. Pointers are then cast to unsigned integers and copied 32-bits at a time.

CONCLUSIONS

In summary, this paper has presented a new architecture for peer-to-peer inter-process communication especially tailored to use with digital human models. Although IPC capabilities are not necessarily novel and applications are increasing, use for integrating complex multi-scale modeling and concurrent design is minimal, and practical use within the DHM community is yet to be exploited. The proposed system uses a TCP/IP protocol for necessary security and for ensuring all data packets are received in order. In general, the system has been designed to facilitate an uninterrupted workflow, which is critical for applications that involve concurrent team-oriented analysis and design. In addition, this new system supports multiple connection mediums and facilitates variations in the number of users. The common issues of blocking and potential lost data have been addressed successfully. Test cases were successfully run to ensure appropriate computational speed.

Implications of the proposed work warrant discussion regarding the fields of digital human modeling and inter-process communication as well. Clearly, the proposed system provides a platform for building a more comprehensive digital human model. However, this mode of development, whereby various teams located in different places contribute individual but potentially interconnected models, is a target of growing significance for many industries. Thus, this work is a significant step towards truly concurrent engineering and development. Along with the growing prevalence of dig data, we anticipate a growth in modeling-and-simulation tools, and just as data must be traversed and coordinated, various related models must be interconnected (regardless of the application or field) in order to maximize effectiveness.

One of the major achievements of the project is continuous, non-blocking operation. This has been a difficult process to understand, particularly when analyzing a user’s work-flow (how they typically use DHM software). The insights gained here are invaluable and have encouraged us to continue collecting this information in the future. We have in effect, designed a networking solution partly based around how one uses DHM software, and the requirements of DHM software in general. This is a unique application of networking design principles. Most network design looks solely at sending and receiving data regardless of the application performing the networking, or the people using that software.

Approaching the problem as Inter Process Communications rather than straight Networking set the design goals. The term IPC leads to a very specific set of goals and thus final product. IPC means communications between applications, not communications between machines, or networking between machines. In the latter mind-set, committing to one specific communications medium such as TCP/IP is not possible. From the outset of the project the code was structured to support multiple communication methods, and to have peers using different methods talk to each other. The current solution supports two diverse methods, in fact completely dissimilar mechanisms. This makes it possible for us to plan and implement new communications medium in the future. This is a rare feature in current communications design, and the proposed architecture reflects the lack of applicable resources in this area.

With respect to potential future work, the IPC library is currently designed as a 32-bit program for maximum implementation flexibility. A 64-bit version is planned. Both the C++ and C# interface can be optimized specifically for 64-bit, including 64-bit copying and marshaling of data blocks. Applications written in 64-bit can also use double the amount of memory (8GB instead of 4GB). Currently, the maximum contiguous packet size that can be sent through the library is 2GB. In a 64-bit version this limit can be doubled. There are no implications involving peers using a mix of 32-bit and 64-bit versions of the library; the transmitted data remains the same.

Also as future work, a WebSockets system will be added. This system works like TCP/IP, in that it is an Internet protocol, but it suffers none of the disadvantages of TCP/IP. It does however, present new problems of its own. TCP/IP requires the peer to have an external IP address and port number to connect to outside peers in external locations. This usually requires an IT technician or network administrator’s input, and is often frowned upon for security reasons. It is then necessary to email or otherwise convey your IP and port number to the other peers you wish to connect to, and in return get their details back. WebSockets solves these problems by communicating through port 1080, the way a web browser does. The communication system resides on a website that is setup by one of the peers. Peers connect to the site and in return to each other. Connection through the web is almost always open and public in every organization; if it were not, there would be no Internet access, and no email. Also, no IP addresses or port numbers are required. The disadvantage, however, is that a client must provide and maintain a private web server that has the WebSockets software installed and running at all times.

This could be a considerable resource to manage for a smaller group.

Many communication libraries include traffic and congestion analysis, and investigating such a system is planned future work. Traffic analysis predicts peers data transmission habits based on their history. Transmission to those peers may delay in order to lesson congestion and thus keep the network running at a constant, albeit slower speed. Congestion analysis is sometimes predictive and in other implementations reactive. This form of network control looks at line quality, or how fast each peer connection is. The result is largely the same as traffic prediction. These mechanisms have been purposely ignored in the initial design. They are used almost exclusively for Intranet and Internet traffic over TCP/IP. An ultimate goal is to support multiple communication mediums and therefore support traffic or congestion prediction that works universally. As a single method for TCP/IP presents considerable work on its own, this development has been left as a future work.

ACKNOWLEDGEMENTS

This work is funded by the Office of Naval Research, under program N00014-13-C-0161, “High Fidelity Digital Human Models for Protective Equipment Design.” The authors would like to thank Mr. Stefan Rublowsky and Mr. Dan O’Donnell, of Crye Associates, for their feedback regarding system use.

REFERENCES

[1] Abdel-Malek K, Arora J, Yang J, Marler T. Digital Warfighter Modeling for Military Applications, in Handbook of Military Industrial Engineering, London, England, Taylor and Francis Press, 2008.

[2] Heart FE, Kahn RE, Ornstein SM, Crowther WR, Walden DC. The Interface Message Processor for the ARPA Computer Network. 1970; p. 551-597.

[3] Kahn RE. Resource-Sharing Computer Communication Networks, in Processing of the IEEE. 1972; vol. 60, no. 11.

[4] Kahn RE. Communications Principles for Operating Systems, Internal BBN memorandum. 1972.

[5] Kleinrock L. Queueing Systems, in Vol II, Computer Applications, New York, John Wiley and Sons. 1976.

[6] Metcalfe MR, Boggs RD. Ethernet: Distributed Packet-Switching For Local Computer Networks, Communications of the ACM, p. Volume 26 Issue 1, 1983.

[7] Grossman L. The Age of Doom, Time Magazine, 2 August 2004.

[8] Sundaresan K, Anantharaman V, Hsieh H, Sivakumar R. ATP: A Reliable Transport Protocol for Ad-hoc Networks, in The ACM International Symposium, Hangzhou, China, 2003.

[9] Gu Y, Hong X, Grossman RL. Experiences in Design and Implementation of a High Performance Transport Protocol, Chicago, IL: Laboratory for Advanced Computing, University of Illinois at Chicago, 2013.

[10] Marler T, Capdevila N, Kersten J, Taylor A, Wanger S, Xie W, et al. “Task-Based Survivability Analysis: an Overview of Capabilities,” 3rd International Digital Human Modeling Symposium, May, Tokyo, Japan 2014.

[11] Marler T, Sultan S. “Multi-scale Human Modeling for Injury Prevention,” 2nd International Conference on Applied Digital Human Modeling, July, San Francisco, CA 2012.

[12] Sultan S, Marler T. “Multi-Scale Predictive Human Model for Preventing Injuries in the Ankle and Knee,” 4rd International Conference on Applied Digital Human Modeling, July, Las Vegas, Nevada 2015.

Received on 30-06-2015 Accepted on 10-07-2015 Published on 18-08-2015

http://dx.doi.org/10.15379/2410-2938.2015.02.01.05