Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics
San Antonio, TX, USA - October 2009

A Software Framework for Multimodal Human-Computer Interaction Systems

Jie Shen
Department of Computing
Imperial College London
London, U.K.
js1907@doc.ic.ac.uk

Maja Pantic
Department of Computing
Imperial College London, U.K.
EEMCS, Univ. Twente, N.L.
maja@doc.ic.ac.uk

***Abstract*—This paper describes a software framework we designed and implemented for the development and research in the area of multimodal human-computer interface. The proposed framework is based on publish / subscribe architecture, which allows developers and researchers to conveniently configure, test and expand their system in a modular and incremental manner. In order to achieve reliable and efficient data transport between modules while still providing a high degree of system flexibility, the framework uses a shared-memory based data transport protocol for message delivery together with a TCP based system management protocol to maintain the integrity of system structure at runtime. The framework is delivered as a communication middleware, providing a basic system manager and well-documented, easy-to-use and open source C++ SDKs supporting both module development and server extension. The experimental comparison between the proposed framework and other similar tools available to the community indicates that our framework greatly outperforms the others in terms of average message latency, maximum data throughput and CPU consumption level, especially in heavy workload scenarios. To demonstrate the performance of our framework in real world applications, we have built a demo system which is used to detect faces and facial feature points in real-time captured video. The result shows our framework is capable of delivering some tens of megabytes of data per second effectively and efficiently even under tight resource constraint.**

***Keywords*—Software Framework, Multimodal Human-Computer Interface, Publish / Subscribe Architecture**

I. INTRODUCTION

With the widely accepted prediction that ubiquitous computing will become the next milestone in the development of computing, multimodal human-computer interface (MHCI) has becoming an active area among the community [2]. Unlike traditional human-computer interface, MHCI is expected to interact with users in a more natural and human-centered way and therefore has great potential for applications in future pervasive systems [2].

While great attention has been attracted by algorithm research in extracting information from different modalities, such as speech, facial expression, gaze, gesture, and so on, the issue regarding software framework, which is essential for turning existing algorithms into reusable modules and subsequently integrating such modules into applicable systems, is largely overlooked. Although many researchers did provide software implementation for their proposed algorithms, these so-called modules, which are normally in OpenCV style, do not have uniform external interface and are highly specific to their own testing / demonstration system [3] [4] [6]. Therefore, it is often hard to reuse a certain algorithm for new projects, especially in cases where the system in mind is expected to be built upon a large number of heterogamous algorithm implementations.

This current situation not only affects projects involving with system integration, but also has negative impact on algorithm investigation since testing these algorithms in realistic and collaborative context becomes difficult. Hence, it is highly necessary to have a software framework which regulates and facilitates rapid module development and system integration for MHCI systems.

A. Requirements to the Framework

An ideal MHCI system is expected to be automatic, adaptive, robust, 'transparent' and responsive [1] – [7]. Hence, the following requirements are considered essential for its underlying software framework:

- Support of heterogeneous modules integration: Algorithms dealing with different modalities may drastically vary in their internal structure. Thus, the framework should not rely on a one-fits-all model for all modules.
- Support of complex system structure: Because feature-level and model-level (rather than decision-level) fusion based approaches are receiving increasing attention [4], individual algorithms are becoming more and more interdependent. With this trend, complex spatial and temporal module relationships within MHCI systems are expected and therefore should be supported by the framework.
- Support of dynamic system structure reconfiguration: Considering that most algorithms only work well in very specific environment, dynamic system structure reconfiguration would be an effective approach

This work has been supported by the European Research Council under the ERC Starting Grant agreement no. ERC-2007-StG-203143 (MAHNOB).

978-1-4244-2794-9/09/$25.00 ©2009 IEEE

towards adaptive and robust performance at system level. For instance, consider a general facial feature point detector (FFPD) which works well for both front view and profile view faces. Complexity of such algorithms is usually much higher than that of specialized detectors (e.g. two FFPDs optimized for each case respectively which are activated / inactivated at runtime with respect to trigger messages sent from a front view face vs. profile view face classifier).

- Guarantee of reliable communication: Data loss may not be that severe to systems with fixed structure where only data messages are transmitted between modules. However, for a system which may reconfigure its structure using triggers, loosing such messages would result in significant performance pitfall. Therefore, it is important that the framework should guarantee successful sending of every message or at least it should notify the sender if the delivery fails.
- Low resource consumption: Because MHCI systems are expected to use as little resource as possible to provide real-time reactivity to users' interactive actions, the framework itself should also keep its resource consumption low. Moreover, it is important for the framework to support compiled modules (in order words, modules written in languages such as C/C++) in order to achieve high overall efficiency.
- Support of large data throughput: Since audio and video signals, which are both high bit-rate streams, are the primary information sources of most modalities under study, the framework should be able to efficiently deliver large amount of data. Even considering the fact that higher level modules often work on abstract information which contains less data, the data flow between a small number of front-end modules can still easily add up to a large value.
- Support of short message latency: Although the maximum allowance for responding time is largely application-dependent, an 'as quick as possible' response is always a goal. Therefore, a long time spent on message delivery is unwanted, especially in large systems where message latencies at each level of the processing cycle will be eventually accumulated.

Additionally, to be of use to a broad community, we also expect the software framework to be well-documented, easy-to-use and BUG-free. Moreover, Open source is another welcomed feature for the customizability it brings.

B. An Overview of Available Tools

Although there are a number of existing tools of the kind we describe here, none of them fulfills all of the aforementioned requirements. Table I provides an overview of the existing tools. These tools can be categorized into two types: SDKs based on local / remote procedure call [8] [11], and middleware based on publish / subscribe (P/S) architecture [9] [10] [12]. The first category is, on one hand, rather intuitive and generally has good performance in terms of data throughput, message latency and resource consumption level. But on the other hand, lack of flexibility is their common drawback. In comparison, the second category has less restriction on the spatial and temporal structure of the system and thus better fulfills the needs of spatially and temporally distributed processing typical of MHCI systems. However, their problem is that these tools are poorly implemented in practice and cannot support high data rate and low latency communication. Therefore, we set out to create a new software framework which meets the outlined requirements in both flexibility and performance.

TABLE I. AN OVERVIEW OF AVAILABLE TOOLS

Software Framework	**Features**			
	Module	***Structure***	***Configuring***	***Reliability***
Microsoft DirectShow [8]	Data stream processor	No feedback loop	Static	Guaranteed
Psyclone AIOS [9]	No restriction	Any structure	Static and dynamic	Message may get lost
ActiveMQ [10]	No restriction	Any structure	Dynamic	Guaranteed
Open-Interface[a] [11]	No restriction	No multicast	Static	
Fleeble[b] [12]	No restriction	Any structure	Static and dynamic	
Software Framework	**Features**			
	Resource Consumption	***Data Throughput***	***Message Latency***	***Documentation Support***
Microsoft DirectShow [8]	Low	Very High	Very Low	Comprehensive and easy to follow
Psyclone AIOS [9]	High[c, d]	< 21 MB/s	Up to 7500+ ms	Easy to follow, but incomplete and inconsistent
ActiveMQ [10]	High[c, d]	< 10 MB/s	Up to 250+ ms	Comprehensive and easy to follow
Open-Interface [11]				Very poor
Fleeble [12]				Easy to follow but incomplete
Software Framework	**Features**			
	BUG	***Supported Languages***	***Open Source?***	***Note***
Microsoft DirectShow [8]	Not detected	C++, C#, VB	No	Module development is relatively hard
Psyclone AIOS [9]	Deadlock, access error and connection failure	C++, Java	No	
ActiveMQ [10]	Memory Leak	C++, Java	Yes	
Open-Interface [11]	No working example	C++, Java, Matlab	Yes	
Fleeble [12]		Java	Yes	

a. OpenInterface was not tested due to its buggy implementation.

b. Fleeble was not tested because it only supports Java and therefore will not be used anyway.

c. Very high CPU consumption rate was recorded when communication workload was heavy.

d. The initial memory consumption was more than 50 MB, which increased rapidly thereafter.

C. Organization of the Paper

The rest of this paper is organized as follows. Section II describes the conceptual design of our proposed software framework, including its P/S architecture and two protocols designed to achieve high data rate communication and flexible

system management, respectively. An introduction to the implementation detail and the delivered tool set is given in Section III. Experimental evaluation of the framework in both artificial stress test settings and real world application settings is described in section IV. And finally, section V concludes the paper.

II. Conceptual Design

This section describes the design of the proposed software framework. We first introduce the publish / subscribe (P/S) architecture used by the framework, followed by the data transport protocol which uses shared memory to facilitate high data rate, low latency and reliable data transport between modules. Subsequently, the TCP based system management protocol, which is used to support dynamic system structure reconfiguration and maintain structure integrity at runtime, is described.

A. Architecture

The proposed software framework is designed and implemented as a middleware facilitating publish / subscribe (P/S) communication between modules.

Fig. 1 illustrates an example. This example system contains three modules. Each of them is built into a standalone executable, which internally calls the client component of our framework to exchange messages with other modules. Different from local / remote procedure call based approaches, in which modules are implemented as components (DLLs, COM objects, and so on), and are called by the framework or other modules, modules in our framework are granted explicit control over their own execution route. In other words, modules do not have to follow any predefined internal structure model as long as they can correctly call the framework's client component whenever communication is needed. In this way, a high degree of flexibility at module level is achieved.

Furthermore, as shown in Fig. 1, modules do not send messages directly to each other, but via logical message dispatchers, which are called channels. Channels are named entities that allow a single message to be dispatched to any number of receivers which have previously shown 'interest' in it [12]. The mechanism behind is as follows. A module informs the framework if it is 'interested' in messages of a certain type by subscribing to the channel dedicated to that type of messages. Then, whenever a message is sent (published) to that channel, the message would be automatically routed to all of the subscribed modules. With this P/S mechanism, modules at both sending and receiving ends are effectively isolated, which means that their dependency on the presence of assumed upstream and / or downstream modules is eliminated. In other words, a module can be used in any circumstance as long as appropriate channels, which are always the same type of entities but with different names, exist. Therefore, development and using of context-free and stateless modules become possible [12] [13].

Additionally, because of this isolation of modules, changes to a part of the system do not affect the whole system, making it possible to build the system in an incremental way, gradually expanding on a small number of interconnected core modules. This way of development, which is called constructionist design methodology (CDM) [13], is suggested to be very appropriate for the implementation of complex and multi-functional interactive intelligent systems, in which detailed specification of the entire system is often unclear at the beginning [13]. This is also normally the case in MHCI research.

With this P/S architecture, the structure of the system is fully defined by the collection of channels and all modules' subscriptions. This decentralized representation of system configuration brings great flexibility to system integration for it does not impose any explicit restriction on the topology of the module network.

Dynamic system reconfiguration is achieved by enabling modules to initiate and / or cancel subscriptions at runtime. In this situation, execution of remaining modules is completely unaffected because each module only sees its input and output channels but not the upstream and downstream modules. Therefore, dynamic system reconfiguration is more or less naturally supported by our framework, without requiring module developers to specifically consider this issue.

Besides modules and channels, a system manager (server) is also required. The role of the system manager is a twofold. Firstly, it hosts all channels. Secondly, the system manager also works as a central repository which stores all information regarding current system configuration at runtime, including a list of channels, a list of working modules, and their subscriptions. Although this central repository is not required in theory, due to the framework's decentralized and implicit representation of system configuration, it is practically essential for the implementation of the P/S communication using our data transport protocol. In fact, our data transport protocol requires each module to carry an identical copy of a part of this information. In order to maintain consistency between all these copies, TCP connection is established between every module and the system manager. A system management protocol is then designed and implemented to synchronize each module's local copy of configuration information with the original copy stored in system manger whenever changes occur. More details on this issue will be given in the next two subsections. Also note that in order to make the system manager extendable, it is actually built into a threaded in-process component called server component (as shown in Fig. 1), which allows developers to add more features into the final executable if necessary.

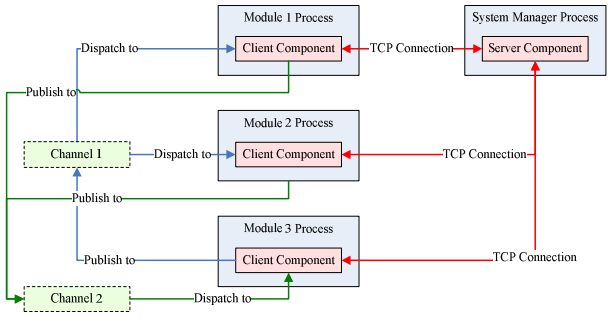

Figure 1. Structure of an example system in which our framework is used.

B. Data Transport

In this subsection, we describe the data transport method used by our framework. Recall that achieving high data rate, low latency, and reliable data transport, which tools like Psyclone and ActiveMQ failed to meet, is one of the most important requirements of a MHCI-supportive software framework.

1) Choosing Inter-Process Communication Method

Because data transport between modules in our framework is basically inter-process communication (IPC), it is important for us to choose a proper underlying IPC method for the protocol in order to fulfill all of aforementioned requirements. The following criteria are crucial when choosing an appropriate method:

- The method should be general enough to support any number of concurrent communication sessions, with messages having arbitrary length. Moreover, it should be able to work in all sorts of programs.
- The method should be reliable enough to guarantee ordered data delivery (first sent, first received).
- The method should be efficient enough, which means it should have the potential to support high data rate and low latency communication. This also means more fundamental methods would be favored to avoid performance overhead.

Although there are 11 different IPC methods for Windows [14] [15] (which is the primary platform on which MHCI systems run [1] [7]), our conceptual evaluation shows that most of them are either too specific (clipboard, WM_COPYDATA and LPC [15]), unreliable (mailslots and UDP) or inefficient (COM, DDE, pipes and RPC). Only TCP and shared memory may meet our requirements.

In practice, TCP is favored by many exiting tools including Psyclone and ActiveMQ due to its convenience of use and its cross-platform nature. However, our experimental comparison between TCP and shared memory, which is given in table II, shows that shared memory can support much higher (up to 10 times) data throughput than TCP under every CPU consumption constraint. Hence, we decided to choose shared memory as the underlying IPC method.

2) Data Transport Protocol

The data transport protocol contains two parts. The first part specifies the peer to peer (PTP) communication protocol directly built upon shared memory and the second half explains how to implement P/S communication using this PTP protocol.

TABLE II. COMPARISON BETWEEN TCP AND SHARED MEMORY

CPU Usage[a]	Data Rate of TCP	Data Rate of Shared Memory
5%	9 MB/s	90 MB/s
20%	34 MB/s	230 MB/s
35%	40 MB/s	245 MB/s
50%	55 MB/s	225 MB/s
65%	73 MB/s	245 MB/s
80%	80 MB/s	280 MB/s

a. Conducted on a ThinkPad T43 laptop with 2.0 GHz Pentium M CPU and 1 GB of memory.

Unlike most IPC methods, shared memory is hardly a communication method. It simply allows developer to create named global memory block, which can be mapped into many processes' address space in order to share data across process boundary [14]. Windows does not provide automatic locking for the memory block to prevent data corruption, but the content of mapped buffer is guaranteed to remain consistent when it is accessed from different processes [14].

Therefore, as the basis of P/S communication, we have firstly defined a protocol to implement reliable PTP communication through shared memory. Below is a rough description of this protocol.

- Each peer is required to allocate a named shared memory block, used as its local inbox. The content of this memory block should be organized as follows. First 4 bytes are used to store an unsigned integer representing the total amount of data currently stored in the inbox, excluding itself. The remaining space is used to store an array of received messages.
- The message format is defined as follows. First 4 bytes store a tag representing application defined message type. Following 16 bytes form an SYSTEMTIME structure storing the sending time of the message. The next arbitrary size of space (at least 1 byte) is used to store a NULL ended string representing sender's name. Then, next 4 bytes are used to store content length of the message as an unsigned integer. And finally, the remaining part of the message stores its content.
- Each peer uses a named mutex to prevent simultaneous multiple accesses to its inbox at any time.
- Each peer should also create a named event used as an indicator representing whether there is data in its local inbox.
- During execution, each peer uses a thread to constantly monitor its indicator event's state. Whenever a set state is detected, the thread would parse its inbox's content (with protected read operations), split the stored data into separate messages, clear the buffer, and then reset the indicator event.
- When a message is sent, the sender simply appends (with protected write operations) the whole message, including both header and its content, into the receiver's inbox and set its indicator event.

To extend this protocol for P/S communication, an intuitive way is to allocate a common inbox for each channel and let the channel's subscribers to constantly monitor its content. However, this method is unreliable. Because there are arbitrarily many subscribers, which may also unsubscribe at any time, it is hard to determine when and whether a message should be removed from the channel buffer, while still ensuring successful delivery to every valid destination. Therefore, we choose to implement the P/S communication using a collection of individual PTP message sending sessions.

In our framework, channel only exists as a logical concept. It consists of only a collection of subscriptions used to guide

message routing. In practice, the system manager stores a channel list. Each of its elements consists of an array containing the name of all the channel's subscribers. Whenever a module needs to publish a message, it simply retrieves the target channel's subscriber list and then sends the message directly to every subscriber through the PTP protocol described above. In this way, message publishing is reduced to a number of PTP message sending sessions, where the reliability is guaranteed. In order to accelerate this procedure, each module also holds a copy of the channel list and therefore eliminates the need of frequent access to the system manager's storage.

C. System Management Protocol

In order to support dynamic system reconfiguration and maintain consistency between the configuration information stored in the system manager and all its copies held by the modules, a system management protocol is designed and implemented. In this protocol, TCP has been chosen as the underlying communication method for its ease of use. Given that the messages used for these purposes are normally short, choosing TCP, which is proved suboptimal in terms of data throughput, should not lead to significant performance degradation.

As stated above, each module establishes and maintains a TCP connection to the system manager during its entire life cycle. This connection is used by the system management protocol for message exchanging. These messages are called system management messages, which are privately used by the framework itself and are transparent to module developers.

In our protocol, 3 types of system management messages are defined:

- Request messages: module registration request, remote channel creation request, remote channel destruction request, module subscription request, and module unsubscription request. These messages are sent from a module to the system manager when the module requests a change in the system structure. The system manager is then required to answer these requests with acknowledgement messages. Note that module logoff request is unnecessary because shutting down the TCP connection carries the same information.
- Notification messages: channel creation notification, channel destruction notification, module subscription notification, and module unsubscription notification. These messages are sent from the system manager to all modules in order to indicate changes in the system structure. Upon receiving, modules should update their local copy of the channel list to reflect the new configuration. Note that there is no module registration notification and module logoff notification because modules are only interested in changes to the channel list. Nevertheless, the system manager is responsible for broadcasting appropriate channel unsubscription notifications whenever a module logs off.
- Acknowledgement messages: ACK (approved) and NACK (rejected) sent by the system manager as the answer to a module's request.

Based on these messages, a number of protocol operations have been defined to handle the situations such as module registration, channel creation, module subscription, and so on. These operations, which are mostly in the typical form of request-process-acknowledge procedure, are straightforward and therefore further description is omitted for brevity.

III. Middleware Implementation and the SDKs

The framework is implemented as a communication middleware, which consists of a basic system manager program, together with well-documented, easy-to-use and open source C++ SDKs for both module development and server extension. In this section, we introduce all these components and briefly describe how they are implemented.

A. Basic System Manager

The basic system manager program is shown in Fig. 2. This program is a shallow GUI encapsulation of the server extension SDK. It displays the channel list, module list and all subscriptions in two tab pages. The program constantly monitors the two lists through the server extension SDK and reflects any change in their content once detected. Through a direct call to the SDK under the hood, users can also create and destroy channels at runtime.

Compared to the equivalent program provided by Psyclone and ActiveMQ's (which are called Psyclone server and ActiveMQ message broker respectively), the function of our basic system manager is rather limited for it supports neither loading / saving of system configuration nor message logging. However, these additional features can be added through programming using our SDKs. For instance, a possible way to implement system wide message logging would be to integrate a logger module (a module which subscribes to all channels and records every message) into the server program.

B. Module Development SDK

Module development SDK encapsulates the implementation of data transport protocol together with the client side of system management protocol and exposes high level programming interface to developers.

The SDK is developed into a C++ static library (.lib). Two precompiled versions are provided, one for Visual Studio 2005 and the other for Visual Studio 6.0. Minor changes may be needed to avoid compiler warnings if the source code is to be compiled for other compiler / linker tool set. Because only standard Windows API and C++ STL library are used during its development, the SDK can be used in any Windows application.

The programming interface is provided through a C++ class called CFEClientComponent, which exports functions for module registration / logoff, module subscription / unsubscription, remote channel creation / destruction and message publishing / receiving. Internal locks are used to avoid data corruption. Therefore, objects of this class can be accessed in parallel from different threads without explicit protection. As the client implementation of the communication middleware, every module should maintain an instance of this class during its entire life cycle.

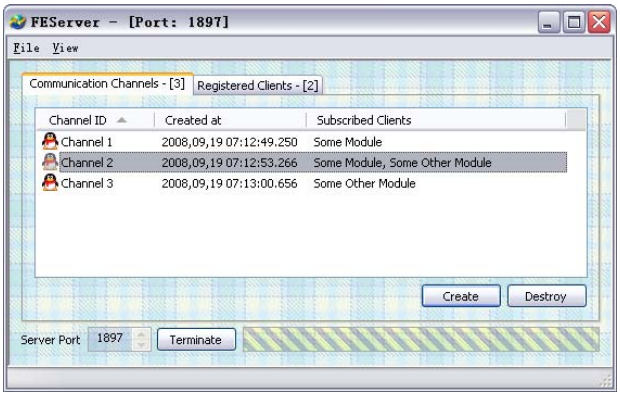

Figure 2. The basic system manager program.

Inside the SDK, three worker threads are used, which are described below.

- TCP communication thread: This thread is used to handle low level TCP communication between the module and the system manager. It continuously monitors the TCP socket, splits incoming data stream into separate system management messages and puts them into an internal queue.
- System management protocol client thread: This thread is used to perform the protocol operations defined in our system management protocol. It parses the buffered system management messages and changes the locally stored channel list accordingly.
- Local inbox monitoring thread: This thread implements the data transport protocol. It monitors the module's local inbox, extracts received data messages and moves them to another internal buffer. These messages would then be returned to the caller when appropriated interface function is called.

C. *Server Extension SDK*

Server extension SDK provides the core implementation of the system manager's functions defined in our framework. Similar to the module development SDK, it is also implemented as a C++ static library which only uses Windows API and C++ STL library, and is precompiled for Visual Studio 2005 and Visual Studio 6.0.

A multithread safe C++ class called CFEServerComponent is provided as the SDK's programming interface. This class exports functions for server startup / termination, channel creation / destruction and system configuration information retrieval.

The easiest way to build an applicable but rather basic system manager program is to create an instance of the CFEServetComponent class as main function's local variable, and call CFEServetComponent::Initialize function to start its internal worker threads when the program starts. More powerful implementations can be obtained through a combined use of both server extension SDK and module development SDK, together with careful programming.

The worker threads used inside the server extension SDK are described below.

- TCP listening thread: This thread is used to monitor the TCP listening socket operated by the system manger. It simply accepts every pending connection request and allocates resource for the newly registered module.
- TCP communication thread: The server allocates one such thread for each registered module to handle low level communication. Similar to its counterpart in module development SDK, the thread is responsible for splitting individual system management messages from the raw TCP stream.
- System management protocol server thread: This thread is used to check buffered system management messages received from all active modules and respond to them according to the protocol operations defined in our system management protocol.

IV. EXPERIMENTAL EVALUATION

In order to obtain performance measure of our proposed framework, we have conducted two experiments. The first one gives a quantitative comparison between our framework, Psyclone and ActiveMQ through stress tests performed in artificial settings. In the second experiment, we have developed a demo system which detects faces and facial feature points in real-time captured video stream. This system not only provides performance measure of our framework in realistic context with tight resource constraint, but also serves as a prototype for further development of a GUI-enabled workbench for our framework.

A. *Comparison to Psyclone and ActiveMQ*

In this experiment, we have developed a test program to compare the performance of our framework, Psyclone and ActiveMQ in terms of maximum data throughput, average message latency and CPU consumption level.

The test program implements a pair of message publisher / subscriber for each dispatcher provided by these three frameworks. When the test is running, the publisher continually sends messages in specified length to the subscriber through the framework's message dispatcher (be it a channel in our framework, a whiteboard / stream in Psyclone or a queue / topic in ActiveMQ). A 10 ms halt takes place between every consecutive message publishing to avoid busy waiting. Then, given a specified message length, its correspondent packet rate, data throughput, average message delay and CPU consumption rate would be estimated and recorded.

In the actual experiment, we ran the test program several times for each framework / dispatcher combination using one publisher and one subscriber. The initial message length was set to 1 KB, which increased to 1024 KB by a factor of 4. Results obtained from this experiment are shown in Table III.

As we can see from the table, for every tested framework / dispatcher combination except of our own, there is a sharp rise in CPU consumption rate, from less than 10% to more than 75%, between some two consecutive rows. Because using more than 75% of CPU power solely for data delivery is clearly out

of question in real-world MHCI applications, a framework / dispatcher combination would be considered intractable after this sharp rise takes place. Therefore, we define the practical maximum data throughput for a combination as its data throughput recorded immediately after this turning point. This value is 6010.88 KB/s for Psyclone / whiteboard, 21268.48 KB/s for Psyclone / stream, 7690.24 KB/s for ActiveMQ / queue and 6969.28 KB/s for ActiveMQ / topic, respectively. In contrast, because a similar increase did not appear when testing our own framework, it is safe to conclude that our framework's practical upper limit in data throughput is at least 102287.36 KB/s, which is already significantly larger than that of any other tested tools.

From another perspective, because in practice, message length is a good indicator of overall communication workload, we can conclude that our framework consumes much less CPU power than the others in heavy workload situations and it does at least equally well when the workload is light.

Regarding to message latency, even if we consider only the value obtained when all combinations were working in their tractable range, our framework already greatly outperforms the others except topic in ActiveMQ, which is still no match to our framework if heavier workload situations are taken into consideration.

B. The Facial Feature Point Detection System

In this experiment, we have developed a demo system in order to measure the performance of our framework in the context of a realistic system with tight resource constraint.

The demo system, which is shown in Fig. 3, is used to detect faces and facial feature points from real-time captured video stream. The data processing is basically performed in a 3-stage waterfall processing cycle. The front-end consists of a video capturer, which captures real-time video frames and pushes them down to the face detector through the original video channel. The output of the second stage is then fed to the facial feature point detector (FFPD) through face detection result channel. The final result, which consists of original video frame together with detected face boxes and facial feature points coordinates, is published to the point detection result channel. Each of the three channels also dispatches messages to a correspondent renderer module, which displays them onto the system's main GUI for visualization.

In this system, both the face detector and FFPD are implemented as standalone executables, which are based on OpenCV's implementation [16] and the algorithm explained in [17].

In order to allow users to freely enable and disable each stage in the processing waterfall, each renderer is also used as the controller and monitor of its 'input' module. They publish status messages to status report channel and respond to command messages received from system command channel. As the abstraction of the main GUI, the system command center module lies in the other end of above two channels and controls the overall execution of the entire system.

In the experiment, we have ran the system 4 times using different input frame size, which implies different level in overall data throughput (roughly 5 times of the source video's data rate). The average message latency and processing time at each detection stage were then estimated and recorded.

The test results are summarized in Table IV. When compared to the result listed in Table III, increase in message latency becomes evident. However, this is also expected since FFPD alone used almost 100% of the CPU power on its own and, therefore, left very limited resource for other processes. Nevertheless, the ratio between total latency and total processing time is less than 0.5‰ in all cases, which indicates that our framework is still able to deliver some tens of megabytes of data per second highly efficient, even in environment where tight resource constraint is imposed.

TABLE III. A QUANTITATIVE COMPARISON BETWEEN OUR FRAMEWORK, PSYCLONE AND ACTIVEMQ

Framework and Dispatcher	Message Length (KB)	Message Rate (baud)	Data Rate (KB/s)	Average Latency (ms)	CPU Usage Rate[a]
Our Framework / Channel	1	99.86	99.86	< 0.01	< 1%
	4	99.86	399.43	< 0.01	< 1%
	16	99.84	1597.44	< 0.01	< 1%
	64	99.84	6389.76	< 0.01	< 1%
	256	99.84	25559.04	< 0.01	< 1%
	1024	99.89	102287.36	0.07	< 1%
Pscylone / Whiteboard	1	98.86	98.86	0.12	< 1%
	4	98.86	395.43	0.13	< 1%
	16	98.56	1576.96	0.14	2%
	64	98.92	6010.88	36.59	100%
	256	47.88	12257.28	2320.69	100%
	1024	14.66	15011.84	7677.53	100%
Psyclone / Stream	1	99.56	99.56	0.04	<1%
	4	99.46	397.83	0.07	<1%
	16	99.84	1597.44	0.11	<1%
	64	99.68	6379.52	0.13	2%
	256	83.08	21268.48	5012.15	90%
	1024	19.36	19824.64	4351.02	100%
ActiveMQ / Queue	1	99.79	99.79	0.01	<1%
	4	99.36	397.43	0.05	<1%
	16	99.84	1597.44	0.02	<1%
	64	93.12	5959.68	0.77	7%
	256	30.04	7690.24	60.32	97%
	1024	7.97	8161.28	277.41	98%
ActiveMQ / Topic	1	99.82	99.82	<0.01	<1%
	4	99.86	399.43	<0.01	<1%
	16	99.84	1597.44	<0.01	<1%
	64	99.84	6389.76	<0.01	2%
	256	24.88	6369.28	30.31	80%
	1024	4.89	5007.36	195.74	98%

a. Conducted on a ThinkPad T42 laptop with 1.7 GHz Pentium M CPU and 512 MB of memory.

TABLE IV. LATENCY AND PROCESSING TIME IN THE DEMO SYSTEM

Frame Size[a]	176 x 144	320 x 240	352 x 288	640 x 480
Frame Rate (FPS)	29.916	29.916	29.916	22.006
Overall Data Throughput (MB/s)	10.85	32.87	43.38	96.71
Average Message Latency (ms)	0.141	0.338	0.490	4.608
Face Detection Time per Frame (ms)	2.469	8.251	11.077	43.878
Point Detection Time per Frame (ms)	11416.667	11538.000	13430.667	18757.000
Total Latency / Total Processing Time	0.024‰	0.058‰	0.073‰	0.490‰

a. Conducted on a HP Mobile Workstation with 2.53 GHz Core Duo CPU and 2.99 GB of memory.

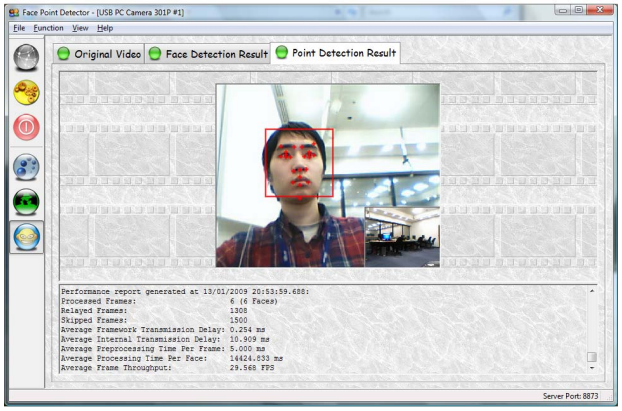

Figure 3. The facial feature point detection system. The buttons on the left-hand-side of the GUI, from top to bottom, are used to: start the system; configure video capturer; stop the system; toggle off both face detector and FFPD; toggle off FFPD only; and toggle on all modules, respectively.

V. CONCLUSION

Based on publish / subscribe architecture, we have designed and implemented a software framework for multimodal human-computer interaction systems. The framework enables rapid development and prototyping on both module and system levels and allows developers and researchers to build their system in a modular and incremental way. It also supports integration of heterogeneous modules and dynamic system structure reconfiguration, which offer great flexibility to system design and implementation. Internally, the framework uses a shared memory based data transport protocol to achieve reliable and efficient message delivery between modules and a TCP based system management protocol to maintain integrity of system configuration at runtime.

The framework is implemented as a communication middleware consisting of a basic system manager program together with well-document, easy-to-use and open source SDKs for both module development and server extension. The SDKs are developed into two C++ static libraries. Each of them provides a multithread safe class as its programming interface. Because only Windows API and C++ STL library are used during their development, both SDKs can be used in any Windows application.

The quantitative comparison between our framework, Psyclone and ActiveMQ in artificial stress test settings shows that our framework greatly outperforms the others in terms of maximum data throughput, message latency and CPU consumption level, especially in heavy workload situations. A facial feature point detection system was also developed as a demo system, which is used to further test the framework's performance in the context of a real world application with tight resource constraint. The result shows that our framework works well in this realistic environment where tight resource constraint is imposed. Though most CPU power is allocated to the facial feature point detection algorithm, the framework can still delivery some tens of megabytes of data per second efficiently and effectively with relatively short transmission latency.

In the next step, using our demo system as the basis, we will build a workbench for our framework, which provides a GUI allowing developers to configure the system structure using simple drag-and-drop operations. Saving and loading the system configuration with disk files will be supported as well.

REFERENCES

[1] R. Sharma, V. I. Pavlovic, and T. S. Huang, "Toward multimodal human–computer interface", proceedings of the IEEE, vol. 86, no. 5, May 1998.

[2] M. Pantic, and L. J. M. Rothkrantz, "Towards an affect-sensitive multimodal human-computer interaction", Proceedings of the IEEE, vol. 91, no. 9, pp. 1370-1390, September 2003.

[3] A. Jaimes, and N. Sebe, "Multimodal human–computer interaction: a survey", Computer Vision and Image Understanding, vol. 108, no.1-2, pp. 116-134, 2007.

[4] M. Pantic, A. Pentland, A. Nijholt and T.S. Huang, "Human computing and machine understanding of human behavior: a survey", Artificial Intelligence For Human Computing, T.S. Huang, A. Nijholt, M. Pantic and A. Pentland, Eds. Springer, Lecture Notes in Artificial Intelligence, vol. 4451, pp. 47-71, 2007.

[5] M. Pantic, A. Nijholt, A. Pentland, and T. Huang, "Human-centred intelligent human-computer interaction (HCI^2): how far are we from attaining it?", Int'l Journal of Autonomous and Adaptive Communications Systems, vol. 1, no. 2, pp. 168-187, 2008.

[6] Z. Zeng, M. Pantic, G.I. Roisman and T.S. Huang, "A survey of affect recognition methods: audio, visual, and spontaneous expressions", IEEE Transactions on Pattern Analysis and Machine Intelligence, 2008.

[7] L. Maat, and M. Pantic, "Gaze-X: adaptive affective multimodal interface for single-user office scenarios", Artificial Intelligence for Human Computing, T. S. Huang, A. Nijholt, M. Pantic, and A. Pentland, Eds. Springer, Lecture Notes in Artificial Intelligence, vol. 4451, pp. 251-271, 2007.

[8] "MSDN: DirectShow (Windows)", Dec. 4, 2008. [Online]. Available: http://msdn.microsoft.com/en-us/library/dd375454(VS.85).aspx [Accessed: Mar. 27, 2009].

[9] "Communicative Machines: Pscylone", 2007. [Online]. Available: http://www.cmlabs.com/psyclone/ [Accessed: Mar. 27, 2009].

[10] "Apache ActiveMQ", 2009. [Online], Available: http://activemq.apache.org/ [Accessed: Mar. 27, 2009].

[11] J-Y. Lawson, J. Vanderdonckt, and B. Macq, "Rapid prototyping of multimodal interactive applications based on off-the-shelf heterogeneous components", Adjunct Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology, pp. 41-42, 2008.

[12] M. Pantic, R.J. Grootjans, and R. Zwitserloot, "Fleeble agent framework for teaching an introductory course in AI", IADIS International Conference Cognition and Exploratory Learning in Digital Age, 2004.

[13] K. R. Thorisson, H. Benko, D. Abramov, A. Arnold, S. Maskey, and A. Vaseekaran, "Constructionist design methodology for interactive intelligences", AI Magazine, vol. 25, no. 4, pp.77-90, 2004.

[14] "MSDN: inter-process communications", Feb. 12, 2009. [Online]. Available:http://msdn.microsoft.com/en-us/library/aa365574(VS.85).aspx [Accessed: Mar. 27, 2009].

[15] P. Dabak, S. Phadke, and M. Borate, "Local procedure call", Undocumented Windows NT, Foster City: M&T Books, 1999, pp. 143-189.

[16] P. Viola, and M. J. Jones, "Robust real-time face detection", International Journal of Computer Vision, vol. 57, no. 2, pp. 137-154, 2004.

[17] D. Vukadinovic, and M. Pantic, "Fully automatic facial feature point detection using Gabor feature based boosted classifiers", 2005 IEEE International Conference on Systems, Man and Cybernetics, Waikoloa, Hawaii, Oct. 10-12, 2005.