Robust Architecture for Human Language Technology

M. Liu, T. Stanley, J. Baca and J. Picone
Center for Advanced Vehicular Systems
Mississippi State University
{liu, stanley, baca, picone}@cavs.msstate.edu

Abstract

Technological advancements in the past few decades have made the spoken language technology systems very complex. This leads to a need for robust, portable and flexible platforms on which these spoken language technologies can be implemented. The development of the DARPA Communicator system was a significant step in this direction and has fueled the design and development of impressive human language technology applications. Its distributed framework has offered numerous benefits to the research community, including reduced prototype development time, sharing of components across sites, and provision of a standard evaluation platform. It has also enabled development of client-server applications with complex inter-process communication between modules. However, this latter feature, though beneficial, introduces complexities which reduce overall system robustness to failure. In addition, the ability to handle multiple users and multiple applications from a common interface is not innately supported. In this paper, we describe our enhancements to the original Communicator architecture to address robustness issues and to support multiple applications simultaneously.

1. Introduction

Early human language technology systems were designed in a monolithic fashion. As these systems became more complex, this design became untenable. This barrier hindered research efforts by smaller labs as they were dominated by labs which had full-fledged systems [1]. There was no standard and flexible environment where the different speech technologies could be evaluated on a common platform. This led to the concept of distributed processing wherein the monolithic structure was decomposed into number of functional components that could interact through a common protocol. This distributed framework was readily accepted by the research community and has been a cornerstone for the advancement in cutting edge human language technology prototype systems. The DARPA Communicator program has been highly successful in implementing this approach [1]. This enabled the research community to build portable and flexible systems that fueled resource sharing and enabled easy debugging of these complex systems.

The plug-and-play capability of the Communicator architecture is well-known for reducing prototype development time by enabling sharing of components across sites, allowing research groups to specialize in specific technologies and share others. The remarkable features of the Communicator architecture proved invaluable in reducing our initial development time. However we also encountered certain vulnerabilities in the architecture during this phase and the need for additional capabilities in the subsequent expansion of our system to include multiple applications. This paper describes design enhancements made to the original Communicator architecture to address these needs, including automated support of multiple multi-user applications through a common interface, improvements to the system's reliability, and enhanced debugging capabilities. We also present measurements of system performance improvements and plans for future development.

2. Original Architecture

Our initial spoken language technology system was built using an early version of the Communicator architecture that was a hub and spoke message-based architecture. The hub acts as the central routing system and the servers communicate with each other through the hub. While initializing the system, all the necessary servers needed for the specific application are started first followed by the hub. The hub reads the hub script which has information on the list of servers, the port numbers it should listen and the i.p. addresses it should contact. It also reads a set of rules which guides it through routing the messages the hub receives. This specific design makes the hub powerful and very flexible. For example, just by changing a few rules the communication can be re-routed to evaluate modules from different labs.

After the initial start up, the hub sends initialization messages to all the servers. These messages are sent in a common format called “frame” which is an attribute-value structure and consists of a name and frame type [3]. These messages can be sent to a server or the hub and usually triggers an action or is in reply for a specific action. Our system is event driven and the initial trigger comes from the user through the user interface. Figure 1 shows the hub and spoke architecture of our dialog system.

During the initial design phase, we experienced communication deadlocks among servers and memory management issues that were difficult to debug. Basic logging mechanisms were provided to address some of these issues, but certain desirable features were not available, such as automated server startup, error-detection and correction. We anticipated such issues would grow in number and complexity as we added multiple multi-user applications.

As an example, the user interface for our system ran as a client program on a laptop with the computational servers running on a workstation. The original architecture serviced multiple users, but required manual server startup, including the port allocation to avoid port conflicts. Further, it required manual detection and correction of server errors by restarting them from the workstation. In either case, startup or error detection, the laptop and workstation may not be in close proximity. Clearly, one solution to the latter problem was to enhance the system robustness to failure. However, no such solution will remove all errors and their potential grows as the number of applications and users increases. It is important, therefore, to also provide graceful error management.

Supporting multiple applications also required a common interface that allows the user to choose from the applications and coordinates inter-process communication with each application server and process. We designed and integrated this enhanced functionality with the server management module as well.

With respect to robustness, the Communicator architecture implicitly allows a strict “handshaking” protocol, but does not require or provide an implementation of such a protocol. We found that implementing and enforcing such a protocol became critical for system robustness as the number and complexity of our applications grew. We also developed debugging tools with corresponding diagnostics and visual displays specific to this protocol.

3. Architectural Enhancements

Our first and most critical need concerned automating server startup, error detection and correction. Secondly, we required a common interface to allow users to select among applications. In addition, the need for robustness to error and improved debugging capabilities were heightened with multiple applications.

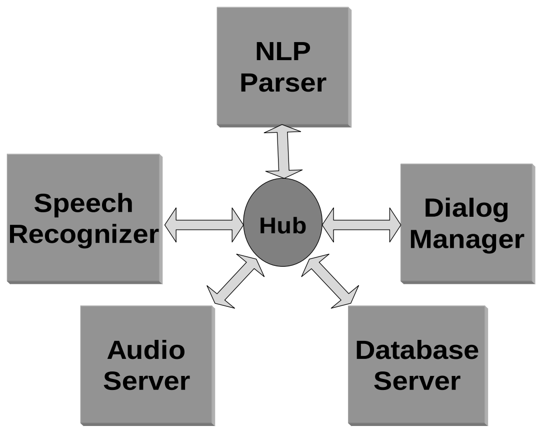


Figure 1. An overview of a DARPA Communicator-based dialog system architecture.

3.1. Automated Server Management

Automated server management became critical with the addition of multiple applications. Though the Communicator process monitor provides an excellent interface to start and terminate servers, it requires manual monitoring. To address this issue, we designed the Process Manager module that automatically starts and controls all server processes in the prototype system architecture. Figure 2 shows an overview of the multi user architecture for multiple applications.

When a user starts a new application, the client program requests the Process Manager to start the respective servers and the hub. The Process Manager performs this startup task by invoking a Java Process Object. The Java Process Object enables the Process Manager module to control all server processes. The Process Manager module can create a process, wait on a process, perform input/output on the process and even check the exit status of the process. If a server process fails for any reason, the Process Manager detects the failure and sends a message to the client side

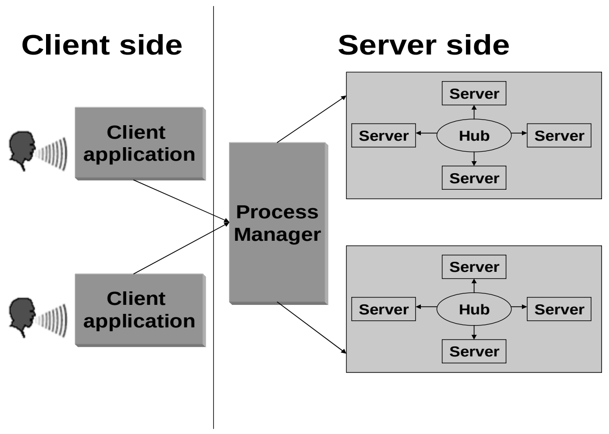


Figure 2. The Process Manager module controls multiple applications and servers.

forcing the user to restart the demo. The Process Manager allocates port numbers and ensures no two servers are assigned the same port.

3.2. Common Application Interface

Support for multiple applications required providing a common interface from which users could select an application of interest. We designed our Demo Selector module to provide the desired interface and coordinate with the Process Manager module to start the required servers.

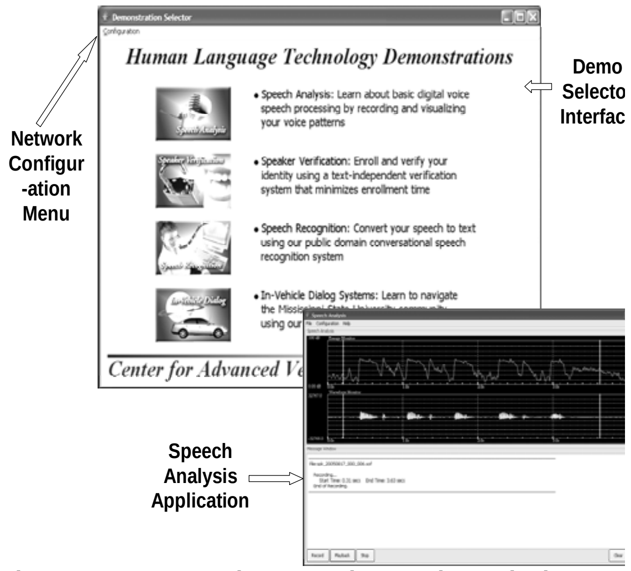
The Demo Selector interface displays a single screen with icons for each of the four applications. Once the user selects an application, the Demo Selector loads and displays the user interface needed for the specific application. Figure 3 shows the Demo Selector interface for the four applications, superimposed with the user interface for the Speech Analysis application, after it has been selected. The client program sends a Communicator frame with a key-value pair containing the name of the application that was selected. Upon receiving the message in this frame, the Process Manager starts the required servers.

3.3. Improvements on System Robustness

Improving system robustness to failure was a primary focus of our enhancements. As the foundation of our redesign strategy, we targeted a simple application, Speech Analysis. Our approach entailed using the implicit capabilities of the Communicator to enhance reliability of inter-process communication between clients and servers. This section describes how we implemented state machine architecture to support a basic handshaking protocol between the client and servers using frames.

Figure 4 shows the state machine architecture and basic handshaking supported between the Speech Analysis client and the Signal Detector server. We used a simple handshaking protocol with signals and acknowledgements, each implemented as Communicator frames sent via the hub. The states and handshaking protocol support three major interaction phases between client and server, 1) preparing for data transfer; 2) data transfer it self, and 3) end of data transfer. For phase 1, the client begins in the Initialization state, during which it establishes connection with the hub. It then transitions to the Audio_Ready state and sends an audio_ready signal to the Signal Detector server to prepare it for audio data transfer. The client then waits for an acknowledgement of the audio_ready signal from the Signal Detector server, and once received, it transitions to the Audio_Ready_Ack state.

Phase 2, data transfer, begins when the client then transitions to the Data_Transfer state and sends packets of audio data in Communicator frames to the server. For each frame of data sent, the client waits for an acknowledgement from the server, which checks each for validity. If the server receives a frame that is invalid, it does not send an acknowledgement signal, but generates an error message, written to a log file. The client will not send further data until it receives an acknowledgement. If data transfer completes successfully, the Signal Detector server detects endpoints and passes the endpointed data to the client. The client then sends an end of utterance signal to the Signal Detector server and waits for an acknowledgement. On receiving the end-of-utterance signal, the Signal Detector server sends an acknowledgement signal to the client and resets itself to the initial state. The handshaking protocol described in this example is implemented for all applications and has eliminated server failures and deadlocks due to communication errors.


Demonstration Selector
Configuration
Human Language Technology Demonstrations
Center for Advanced Ve
Network Configur -ation Menu
Demo Selecto Interfac
Speech Analysis Application

• Speech Analysis: Learn about basic digital voice speech processing by recording and visualizing your voice patterns

• Speaker Verification: Enroll and verify your identity using a text-independent verification system that minimizes enrollment time

• Speech Recognition: Convert your speech to text using our public domain conversational speech recognition system

• In-Vehicle Dialog Systems: Learn to navigate the Mississippi State University community using our

Figure 3. Demo Selector and Speech Analysis user interface

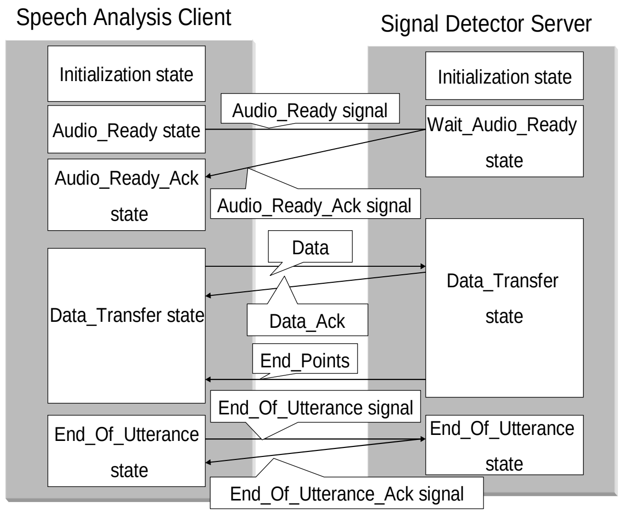


Figure 4. Handshaking between the Speech Analysis client and the Signal Detector server

Table 1. Performance data for a dialog application

		Before Enhancements			After Enhancements		
			Failed (%)			Failed (%)	
Queries	# of utterances	Passed (%)	Server Errors	Deadlocks	Passed (%)	Server Errors	Deadlocks
Address	98	100	0.00	0.00	100.00	0.00	0.00
Direction	219	95.43	2.28	2.28	100.00	0.00	0.00
Distance	23	91.31	8.70	0.00	100.00	0.00	0.00
List of places	36	100	0.00	100	100.00	0.00	0.00
Building	10	100	0.00	100	100.00	0.00	0.00
TOTAL	386	96.92	1.80	1.29	100.00	0.00	0.00

4. Performance Improvements

Table 1 shows the performance data for 386 queries spanning through five different query types for our application. This data was gathered early in our development efforts, prior to our enhancements. Out of the 386 queries, 96.92% “passed” while 3 % “failed” due to a server error or a deadlock. These 386 queries where re-run after these architectural enhancements, and we were successfully able to eliminate these failures related to the robustness of the system.

The state machine architecture enhancements have eliminated fatal server errors and trapped system deadlocks that are due to inter process communication errors. Previously, an invalid data frame sent by the client to the Signal Detector server could potentially cause it to fail. The state machine architecture will prevent such an error: if the client sends an invalid data frame to the server, the server will generate an error message to a log file, and equally importantly, will not send an acknowledgement to the client of the invalid data. Until it receives this acknowledgement, the client will stop data transfer. This also traps a potential deadlock, since the client can be programmed to time out after a specified wait time for an acknowledgement. Further, the state machine debug information written to the log file by the server can be used to isolate and successfully debug where the data transfer error occurred.

Finally, the process monitor of the original architecture could not detect server failures, regardless of their origins. Our enhancements have eliminated one very common cause of server errors. However, if a server fails due to other types of errors, the Process Manager detects the server failure, terminates all servers, and informs the user to restart the system, thus providing a more graceful level of error handling.

5. Conclusions

The DARPA Communicator architecture significantly advanced human language technology and, has played a critical role in the design and development of human language technology applications in our laboratory. In developing these applications, we have addressed vulnerabilities in this architecture through several important enhancements, including automated server startup, error detection and correction, support for multiple multi-user applications, increased system robustness to failure, and improved debugging capabilities.

We also plan to enhance the Process Manager to create and manage server processes on different host machines which will enable us to tap the full potential of our supercomputer clusters.

6. References

[1] K. Hacioglu and B. Pellom, “A Distributed Architecture for Robust Automatic Speech Recognition,” Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 1234-1234, Hong Kong, April 2003.

[2] J. Baca, F. Zheng, H. Gao and J. Picone, “Dialog Systems for Automotive Environments,” Proceedings of the European Conference on Speech Communication and Technology (EUROSPEECH), Geneva, Switzerland, pp. 1929-1932, September 2003.

[3] J. Aberdeen, B. George and S. Bayer, “Galaxy Communicator,” SourceForge.net, Open Source Technology Group, VA Software, Fremont, California, December 2005, (*http://sourceforge.net/projects/communicator*).

[4] W. Ward and B. Pellom, “The CU Communicator System,” Proceedings of the IEEE Automatic Speech Recognition and Understanding Workshop, Keystone, Colorado, USA, pp. 1234-1234, December 1999.