Protocol-Based Interaction in Component-Based Software Systems

Saleh Alhazbi
Computer Science & Engineering Dept.
Qatar University
salhazbi@qu.edu.qa

Aman Jantan
School Of Computer Science,
Universiti Sains Malaysia
11800, Penang, Malaysia
aman@ cs. usm.my

Abstract
In Component-based development, assembling components into systems is the major activity. Therefore, Components must be integrated through well-defined infrastructure. This paper presents a framework for composing component-based systems based on message-pattern interaction among the components; it also presents protocol-based rules to govern messages exchanges.

1. Introduction
Nowadays, component-based paradigm for developing software systems is getting more attention as a methodology for managing complexity in current software systems with more maintainability, adaptability and reliability. In component-oriented model, the whole software system is built by integrating pre-built, pre-tested components rather than implementing every part from scratch. These pre-built components might be developed locally or purchased from a third party (commercial off-the-shelf components (COTS)).
While there are many potential benefits of using component-based model to develop software, it also has some difficulties; the principal problem is how to wire components together. It is no longer sufficient that components just be integratable. They must be interoperable. Interoperability can be defined as the ability of two or more components to communicate and cooperate together to provide system functionalities [1]. Interoperability problems are of two kinds: interface mismatches and protocol mismatches. An interface describes a component's characteristics, e.g., its functionality, structure and performance. A protocol describes the connections the components use for communication [2].
In this paper, we present a Protocol-based Interaction Component-based framework (PICS), which is a framework for composing component-based software systems. In PICS, components communicate by exchanging messages through soft system bus. A predefined protocol governs the way of sending and receiving those messages. This includes messages types, formats, and rules that specify this style of interaction. The main contribution of our approach includes the following:-

- We separate computational part (components) from communication (connectors) to increase reusability and improve system's maintainability. Moreover, such separation supports dynamic changes in system's connectivity [3].
- Another level of separation is provided, where components only talk to connectors (first level), which communicate indirectly through soft bus (second level).
- We provide an xml-based description of component interface that describes not only the services provided by the component but also those are required from other ones during execution [4].
- Our framework could be a step toward standardization for components interaction to create fully plug-and-play software component like that with hardware parts.

This paper is organized as follows. Section 2 presents background of components interactions patterns in general and more specifically on message-based style. Section 3 presents the proposed framework, PICS. Section 4 describes a prototype example and an experience to investigate performance overhead with PICS. Section 5 provides related work. Conclusions and future work directions are given in section 6.

2.Component Interactions
In component-based software systems, the functionalities are not performed within one component; it is done by interacting, cooperating between system's components. Usually, a group of components depend on each other to perform a complex functionality of the system [5]. Dependencies between components can be defined as the reliance of a component on other(s) to support a specific functionality or configuration.

Assembling a system composed of reusable components can be achieved through different patterns to specify components communications style. These patterns include consumer-produces, component glue, and message-based through component bus [6]. From architecture prospective, interaction between components can be achieved either implicitly or via connectors [7]. Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions. Unlike components, connectors might not correspond to compilation units in implemented systems [8]. Explicit connectors also make the bindings between components more loosely coupled; as a result, it increases reusability and reduces dependencies among components which supports faster and better component evolution [9].

2.1 Message-Based Interaction

In message-based interaction style, components communicate with each other by sending and receiving messages. The components of the systems are hooked together to one special component which represents the bus for routing the messages between the components (Figure 1).

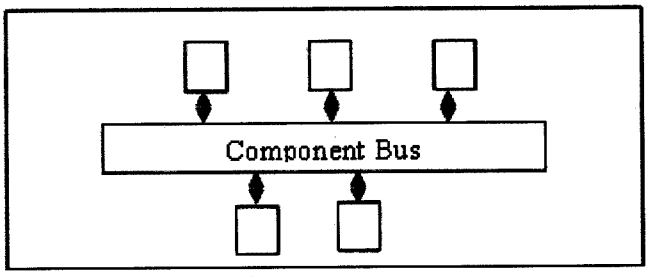

Figure 1

Passing messages between systems' components can be either synchronous or asynchronous. In synchronous style, the sender is blocked until the message is received by the receiver. On the other side, in asynchronous style, the sender sends a message and continues regardless of whether the message has been received or not [10]. From another view, we can distinguish between two approaches of messaging:

- point-to-point where each message is addressed to a specific component,
- publish-subscribe where each message might have multiple receivers.

Generally message-oriented pattern of interaction has the following advantages:

1- All dependencies are centralized and no explicit decencies between components which makes component integration easier [8].
2- It reduces the architecture complexity of the system which means it's more maintainable and adaptable [11,12]
3- Message-based systems are more upgradeable and reconfigurable as new components can be added for satisfying new requirements without changing the basic system architecture[cheng].

However, such a style needs to define an interaction protocol that not only specify interface required for components to interact, but also specifies all rules, formats, and procedures that have been agreed upon between components [13].

3. Proposed Framework

In this section, we present the architecture of PICS and the proposed protocol that defines the interaction between the components. Our framework is based on message interaction style between components. Components send/receive messages through a soft bus to provide the functionalities of the system. Additionally, each component is hooked to the soft bus through a connector to facilitate message exchanges. Figure 2 depicts PICS architecture.

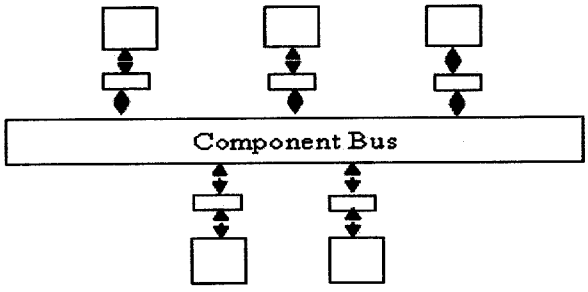

Figure 2

3.1 Components

Generally, in component-based development (CBD), component is defined as "a unit of composition with contractually specified interfaces and explicitly context dependencies only [14]". In our framework, components are the locus of computation. They are service providers and consumers. They cooperate to provide system's functionalities. Any tow components can only communicate if they are syntactically compatible. Compatibility can be described as the ability of two objects to work properly together if connected, i.e. that all exchanged messages and data between them are understood by each other [15]. In our framework, each component has an xml description file that describes its interface; this description includes services provided/required by the component and signature of each service. Figure 3 illustrates an

```
component>
 <name>comp2 </name>
 <provide>
   <service>
         <name>add</name>
          <return>int</return>
          <arg>int</arg>
          <arg>int</arg>
     </service>
</provide>
<required>
   <service>
       <name> getNo</name>
       <return>int  </return>
 </required>
</component>
```

Figure 3

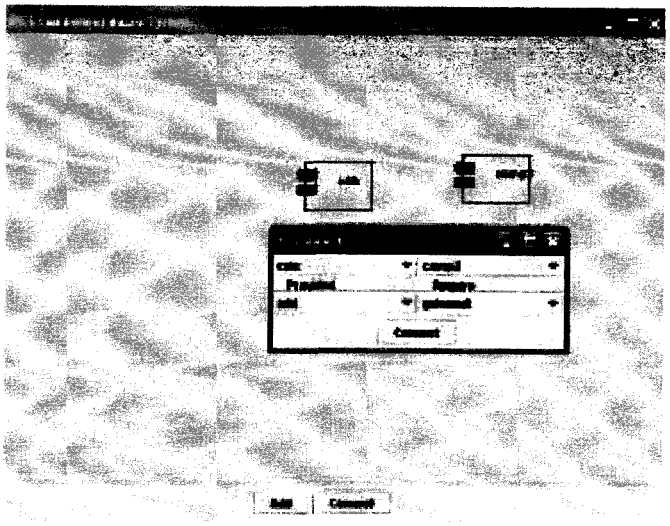

Figure 4

example of xml file to describe a component *comp2* that provides a service *add* which returns an *integer* and has two parameters of type integer. This component requires a service *getNo* that returns integer value and has no arguments.
During integration phase, this description file is used by a visual tool (Figure 4) to wire the components. This java-based tool extracts that component meta-data from the xml, and according to that checks the syntax compatibility between components' interfaces. This visual tool helps generate connectors to hook up the components to the soft bus.

3.2 Connectors

Connectors in our framework are not computation parts of the system, they facilitate components interaction. Each component in PICS communicates with other components in the system through a connector which hooks the component up to the bus. Each connector represents the gateway between the component and the bus. We have two types of connectors Out-port and In-port, Out-port connector masks the services provided by the component, therefore this connector has the same methods as the component behind it. The task of this type is to interpret incoming messages according to the protocol and call the service from the component. Those connectors have the ability to buffer incoming messages when the component is busy. On the other hand, out-port connector represents the gateway for the service required by the component, its task is to set-up outgoing messages according to the protocol. Besides those tasks, both types adapt incompatible messages according to information provided by the designer during integration and supported by our tool "*ComWiring*" (Figure 4).

3.3 Interaction Protocol

Basically, a protocol is a convention or standard that controls or enables the connection, communication, and data transfer between two computing endpoints. In its simplest form, a protocol can be defined as the rules governing the syntax, semantics, and synchronization of communication [16]. The purpose of PICS protocol is to specify communication style, format, and rules between system components. Components communicate each other asking for services or providing results. This interaction in PICS is message-based where we assume there is no duplicated or loses messages.

3.3.1 Messages

Our protocol defines three types of messages: Request message (RQ), Response message (RS), and Failure message (FM). Every message contains two parts: a message part (such as service required, service arguments), and a control part (such as message ID, message type).

1) Request message (RQ): this message is sent from a component to another asking for one of its provided services. The message is six tuple < Message type, Receiver, Service, no of arguments, arguments, sender>
2) Response Message(RS): this message is sent as a successful response to a previous request, it carries the result back to the sender of the request. This message format is five tuple <Message type, Receiver, Result, Sender>, even thought the

service might not returns any result, an RS message should send back to the requester component. RS considered as acknowledgment message of finishing the process.

3)Failure Message (FM): this message is sent as a unsuccessful response to a previous request. Mainly this message is sent back to a component because of runtime error. This message is four tuple <Message type, Receiver, Error, Sender>.

3.3.2 Procedure Rules

The procedure rules of our protocol are described as follows:

a. During runtime, each component has two states: busy, or ready.
b. The message interaction in PICS is synchronous, which means when a component sends a request message, it enters to busy state waiting for RS or FM.
c. When a component receives an RM while it is in busy state, the message is buffered.
d. RM messages are buffered during component is busy as first-in first-out(FIFO).

The flowchart in Figure 5 shows how a component responses to RQ messages.

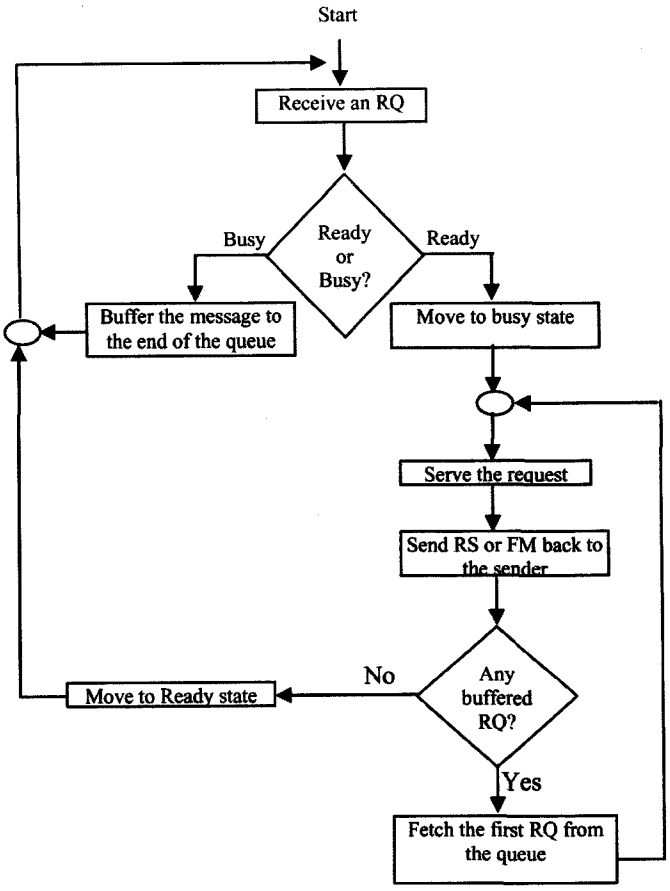

Figure 5

4.Case Study

In this section we present an example as a prototype of PICS framework

4.1 Implementation

We have prototyped our framework in java language where the main component is a class. Figure 6(a) illustrates the implementation of Bus concept in java. Figure 6(b) shows a fragment of the code for MessageEvent which is the super class for both RequestMessage and ResponseMessage classes. Each time a component sends a message, it notifies all components there is a message on the bus, then each component check if it's the destination for that message according to rec_id.

```
public class Bus {
private ArrayList components=new ArrayList();
public synchronized void connectComponent( Reciever l )
{   components.add( l ); }
public synchronized void disConnectComponent( Reciever l )
 {   components.remove( l ); }
public synchronized void sendMessageEvent(MessageEvent m) {
      Iterator listeners = components.iterator();
   while( listeners.hasNext() )
{     ( (Reciever) listeners.next() ).messageRecieved( m );  }
}
}
```

(a)

```
public class MessageEvent extends EventObject {
public MessageEvent(Object arg0,int rec_id,int
send_id) {
        super(arg0);
        receiverId=rec_id;
        senderId=send_id;
}
```

(b)

Figure 6

4.2 Performance Issues

As an example application, we have built a simple application that generates random numbers and use binary search algorithm to look for a specific element in that array. Our example composed of three components: *Gen_Com* to generate an array of random integers, *Sort_Com* to sort the array, and *Search_Com* to search the sorted array. To investigate if there is any overhead performance which may result of using message-based interaction, we built two versions of this application, one using procedure-call as usual in object-oriented mode, and the second version is built based on our framework. In order to get an average time, we run an experiment 50,000 times to generate 10,000 integers each time, and calculate the average time. To keep the time for search algorithm fixed, we always search for the first number in the array. This experiment was run on *3 GHz P4* system with Java version 1.5.0_05. The result shows that there is no difference between the two versions' performance.

6. Related Work

Message-based communication in component-based system offers clear separation and more loosely coupling comparing to other interaction styles such as procedure call, or shared memory. In this section we compare our approach with some relevant approaches. Similar to our approach, C2 architecture [17, 18] uses also message-based interaction between components. On the contrary to ours, C2 uses connectors themselves to facilitate components communication. Therefore, our framework has one more level of separation which decouples rules of interaction from the bus that is only represents the channel to deliver the message. Moreover, in our framework, components send and receive messages synchronously which is simpler style for system designers and integrators as they do not need to worry about deadlock case. In our framework, deadlock case is easy to be found out like the infinite recursive in procedure call style.

Regarding the description of component interface, while the Interface Definition Languages (IDLs) mostly describe only the provided services of a component, our xml-based description here provides description for both provided and required services which is necessary for auto-integration of the components supported by our tool "*ComWiring*".

7. Conclusion and Future Work

In this paper, we present a framework (PICS) for component integration based on pre-defined protocol that governs message exchange among system's components. The concept presented here through PICS is preliminary step toward fully pluggable components for building component-based systems with more maintainability. Moreover this framework supports run-time updating as components can be plug in and out easily to the bus that routs messages among the components. Future work is needed to enhance our tool "*ComWiring*" to ease integration of components. Regarding performance issues, maybe more cases with more components is needed to investigate the impact of the number of components on system performance. Another needed direction for future work is to extend PICS to support transfer state between components during run-time updating.

Reference:

[1] Wegner, P., Interoperablity , ACM Computing Surveys 28(1):285-287.

[2]Vernon M., Lazowska E., and Personic S, editors , R &D for the NII: Technical Challenges. Interuniversity Communications Counicl, Inc. (EDICOM), 1994.

[3] Smeda A., Khammaci T., and Oussalah M., Improving Component-Based Software Architecture by Separating Computations from Interactions, in proceeding of first Internationl Workshop on Coordination and Adaptation Techniques for Softwar Entities (WCAT04), June 2004, Morway.

[4] Olafsson A., and Bryan D., On the need for "required interfaces" of components. In Special Issues in Object-Oriented Programming. Workshop Reader of ECOOP'96, pages 159–165. Dpunkt Verlag, 1996

[5] Vieira M. , et al. "Describing Dependency at Component Access Point", Proc. of Work-shop on Component-based Software Engineering (at ICSE 2001), Toronto, Canada, May 2001.

[6] P.Eskelin, "Component Interaction Patterns", on line Proc,6th Annual Conference on the pattern languages of programs(Plop99) 1999.

[7] Balek, D., and Plasil, F. Software connectors and their role in component deployment. In Proceedings of the IFIP TC6 / WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems (2001), Kluwer, B.V., pp. 69{84}.

[8] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In M. Jazayeri and H. Schauer, editors, Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97), pages 60–76. Springer–Verlag, 1997.

[9] Tansalarak N. and Claypool K., CoCo: Composition Model and Composition Model Implementation In the 7th International Conference on Enterprise Information Systems. (May 24-28, 2005)
[10] Shangzhu Wang, George S. Avrunin, and Lori A. Clarke. In Ian Gorton, George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens A. Szyperski, and Kurt Wallnau, editors, Architectural building blocks for plug-and-play system design, *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2006)*, number 4063 in LNCS, pages 98--113, Västerås, Sweden, June 2006.
[11] Alhazbi, S., Measuring the complexity of component-based system architecture. in: Proceedings. 2004 International Conference on Information and Communication Technologies: From Theory to Applications, 2004., 593-594
[12] Cheng J., "Soft System Bus as a Future Software Technology," Proc. 8th International Symposium on Future Software Technology, Xi'an, China, SEA, October 2004.
[13] Holzmann G. , Design and Validation of Computer Protocols, Prentice Hall in November 1990
[14] C.Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley,1999
[15] Vallecillo A, Herandez J, and Troya J., Component Interoperability, Tech. Rep. ITI-2000-37, Dept. de lenguajes Ciencias de la computación, University of Málaga, July 2000.
[16] http://en.wikipedia.org
[17] Taylor R. , Medvidovic N., Anderson K., E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D.L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. IEEE Transactions on Software Engineering, vol. 22, no. 6, pages 390-406 (June 1996)
[18] Medvidovic N. , Oreizy P., and Taylor R., "Reuse of Off-the-Shelf Components in C2-Style Architectures." In Proceedings of the 1997 Symposium on Software Reusability (SSR'97), pages 190-198, Boston, MA, May 17-19, 1997