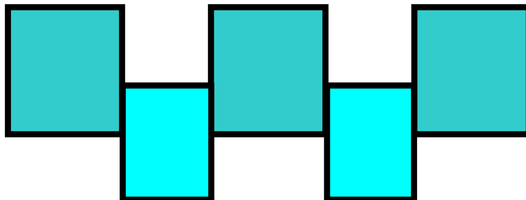
Authors: Peter Poller Date: September 2002

Deliverable 1.1: Review of the state of the art in system architectures

Public document



Document History

Version	Editor	Date	Explanation	Status
0.1	Peter Poller	03.05.02	WP1 state-of-the-art	DRAFT
0.2	Peter Poller	12.08.02	WP1 state-of-the-art	DRAFT
1.0	Peter Poller	27.09.02	PCC approval	FINAL

COMIC

Information sheet issued with Deliverable *D1.1*

Title:	Review of the state of the art in system architectures
Abstract:	This document gives an overview of the state of the art in system architectures suitable for distributed multimodal systems such as COMIC.
Author(s):	Peter Poller
Reviewers:	
Project:	COMIC
Project number:	IST- 2001-32311
Date:	12.08.2002
	For Public Use
Key Words:	Multimodal systems, system architectures, distributed processing
Distribution List:	
COMIC partners	MPI Nijmegen, KUN, DFKI, MPI Tübingen, ViSoft, UEDIN, USHFD
External COMIC	IST, public web site

The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Contents

Summary **7**

1 State of the art in system architectures **9**

1.1 Galaxy Communicator: **10**

1.2 Open Agent Architecture (OAA) **12**

1.3 Pool Architecture (PA) **13**

1.3.1 The Blackboard Metaphor 13
1.3.2 The Module Manager 14
1.3.3 The Testbed: Framework for a Distributed Environment 14

2 COMIC-specific system architecture comparison **17**

3 Conclusions & Recommendations **21**

4 References **23**

Summary

This document presents an overview of relevant system architectures for the multimodal distributed dialog system of the COMIC project. In a first step the various existing communication frameworks based on different communication and messaging models were critically appraised in order to preselect powerful, flexible, appropriate and easy-to-use system architecture candidates for the project. The remaining architectures were then compared within the COMIC Workpackage 1 "System Architecture and Integration" during the first six months of the project (T1 – T6, March 2002 - September 2002). Three different architectures (based on different communication models) emerged as candidates: the Open Agent Architecture (OAA), the Galaxy Communicator and the Pool Communication Architecture (PA). This comparison is intended to prepare the final selection of a system architecture for the project. This document presents first the preselection step, then the main characteristics of these three architectures and then presents the comparison with a special focus on the project-specific requirements.

1 State of the art in system architectures

The COMIC system is a distributed, multimodal, multi-component system. Therefore COMIC needs a software architecture that enables and supports agile, flexible and dynamic composition of the modules and hides as many details of the communication protocol as possible from the module programmers.

This section gives a short overview of the major distributed communication frameworks that can be used to spread interacting components of a large system across multiple computers to achieve distributed processing.

First of all, there are very simple and elementary communication frameworks in which the interactions between components are preconfigured (hard-coded). All details of communication protocols have to be maintained by the programmers themselves. An example of such a hard-coded communication model is Sun's Jini [8]. It realizes just a simple infrastructure to establish a connection over a network.

There are more sophisticated and comfortable distributed models which include an additional communication layer, the so called "middleware", which offers a more powerful mechanism to create distributed applications based on "meaningful" communication objects and hiding some of the more complex protocol tasks from the programmers. Examples are the Common Object Request Broker Architecture (CORBA) [7] or Microsoft's Distributed Component Object Model (DCOM) [6]. A disadvantage of these communication frameworks is that they still need fixed point-to-point communication in the sense that communication partners have to know each other.

The format of the messages being communicated is also a relevant point to look at. E.g., in messages based on the Knowledge Query and Manipulation Language (KQML) [4] intended speech acts can be expressed and arbitrary content can be embedded into textual messages. Unfortunately the models described above also suffer from the necessity of hard-coded point-to-point interactions among components.

Finally, there are communication architectures including a "middleware" layer in which a component can publish requests without any knowledge about the component that fulfils this request. In such architectures data producers are decoupled from data consumers. There are two communication models that fulfil this requirement: the Blackboard Metaphor and the Delegated Computing Model in which a central message brokering unit is responsible for maintenance, coordination, and delivery of requests and their responses to the correct recipients.

The blackboard approach is a widely used architecture and consists of three parts: (i) a set of independent modules, called knowledge sources, which provide the expertise needed to solve the problem; (ii) a blackboard, which is a shared global database which the modules send messages to and receive messages from; and (iii) a control component, which makes runtime decisions about the resource allocation and the order in which the modules operate on the blackboard. At the beginning, each module informs the blackboard about the kind of events it is interested in and contributes with its expertise to the solution of the overall problem. This leads to an incremental solution of the problem. The entire communication between the modules takes place solely through the blackboard. This database encompasses the problem-solving state data, e.g., the hypotheses, the partial solutions, and, more generally, all data exchanged between the modules. Moreover, there are also further developments of the blackboard concept, such as decentralized or parallel extensions of the blackboard architecture.

In a framework based on the Delegated Computing Model there is a specialized central brokering unit that coordinates the activities of the individual modules. At the beginning, each module informs the central broker about its capabilities. Service request messages can then be published to the broker whose task it is to maintain their processing, i.e., the delegation of the request to the recipient(s), the coordination of their efforts and the delivery of the result(s) to the requester.

Architectures that are based on these elaborated computing models do not only contain the necessary communication infrastructure. They are complete integration platforms that significantly support the complex process of integrating a new module into an overall system of communicating modules. Furthermore these architectures include several helpful tools, e.g. a graphical user interface (GUI), methods for easy debugging, and monitoring/logging tools. They are sophisticated architectures that have been successfully used in several complex systems.

In COMIC, such an easy-to-use architecture of either of these two kinds is needed because it hides time-consuming interface and communication programming from the module developers to a significant degree. Only frameworks of this kind are usable in COMIC because they offer the most comfortable and flexible communication and allow module developers to focus on their module functionality instead of loosing time for intermodular interface programming. Finally, they must be freely available either as open source or by one of the project partners.

Consequently, only the following architectures of this kind are relevant for COMIC:

- Galaxy Communicator
- Open Agent Architecture (OAA)
- Pool Architecture (PA)

The following subsections give a short introduction into the main characteristics of the mentioned architectures.

1.1 Galaxy Communicator:

The **Galaxy Communicator** [1] is an open source system architecture based on the plug-and-play approach that enables developers to combine architecture-compliant commercial software and research components. It is a distributed, message-based, hub-and-spoke infrastructure that was especially optimised for spoken dialog systems. The Communicator infrastructure is an extension and evolution of the MIT Galaxy System, and is being developed, maintained, and provided by the MITRE Corporation. The current version 4.0 contains API's for C, C++, Java, Python and Allegro Common Lisp. It runs under Sparc Solaris, Intel Linux and Win32 (x86-NT).

An instance of a Communicator infrastructure consists of arbitrary many processes that may be running on separate computers. The processes are arranged in a hub-and-spoke configuration, which means that a central processing unit (the hub) is used to mediate message based connections between communicating servers/modules (the spokes). The figure below shows a skeleton of a Galaxy Communicator based spoken dialog system:

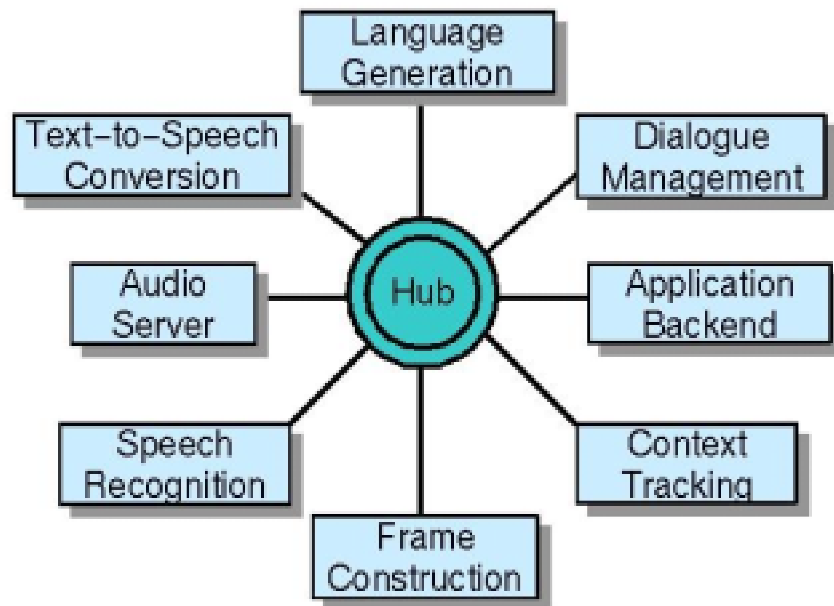


Figure 1: Example Galaxy Communicator instance for a spoken dialog system

The Galaxy Communicator distribution actively supports the platforms Sparc Solaris, Intel Linux and Win32 and consists of:

- A hub, implemented in C, which mediates connections between Communicator servers (such as speech recognition and synthesis, parsing, dialogue management, etc.)
- Server libraries for constructing Communicator-compliant servers in C (and C++), Java, Python, and Allegro Common Lisp
- Examples illustrating basic and advanced functionality for creating servers and setting up the hub to communicate with them
- Extensive documentation
- Example servers such as wrappers for the TrueTalk and Festival speech synthesizers, and for Oracle and PostGres database clients

The infrastructure and especially the hub offers several helpful features/functionalities/tools that ease the development and implementation of a complex Communicator based dialog system:

- The hub explicitly contains message-logging support.
- The message traffic routing can be explicitly programmed by special hub scripts.
- There is a possibility for breakpoint triggering via the so-called “BuiltIn” server, which is a hub internal server that has special access to the internals of hub operations.
- The server libraries support backchannel connections for high-bandwidth data that can be passed directly from server to server.

1.2 Open Agent Architecture (OAA)

The **Open Agent Architecture (OAA)** [5] is a framework for integrating a community of heterogeneous software agents in a distributed environment. The architecture is developed and maintained by SRI, Menlo Park, CA. The architecture runs under Unix, Linux and Windows. It contains API's for C, (Visual) C++, PROLOG and Java.

In OAA an *agent* is defined as any software process that meets the conventions of the conventions of OAA. A key distinguishing feature of OAA is its delegated computing model that enables both human users and software agents to express their requests in terms of *what* is to be done without requiring a specification of *who* is to do the work or *how* it should be performed. Similar to the Galaxy Communicator there is also a central processing unit, the so called facilitator (programmed in PROLOG), which is a specialized server agent within OAA that coordinates the activities of agents for the purpose of achieving higher-level, often complex problem-solving objectives. The knowledge necessary to meet these objectives is distributed in four locations in OAA:

- the requester: It specifies a goal to the facilitator and provides advice on how it should be met,
- providers: They register their capabilities with the facilitator, know what services they can provide, and understand limits on their ability to do so,
- the facilitator: It maintains a list of available provider agents and a set of general strategies for meeting goals
- meta-agents: They contain domain- or goal-specific knowledge and strategies that are used as an aid by the facilitator

Based on this knowledge, the facilitator matches a request to one or more agents providing that service, delegates the task to them, coordinates their efforts, and delivers the results to the requester.

All communication and cooperation between modules is achieved via messages expressed in the Common “universal” Interagent Communication Language (ICL). ICL includes a conversation layer and content layer. The conversational layer is defined by event types together with the parameter lists associated with them.

On the other hand the content layer consists of the specific goals, triggers and data elements that may be embedded within various events. The content layer of ICL has been designed as an extension of the PROLOG programming language to take advantage of unification and other PROLOG features. Thus, compound goals can be expressed by using PROLOG-like operators and also parallel goals to be processed competitively or simultaneously by different modules. Furthermore conditional execution and constraints on executions can also be expressed. Altogether, OAA's ICL offers a very powerful communication mechanism.

The following figure shows an example dialog system embedded into an OAA community.

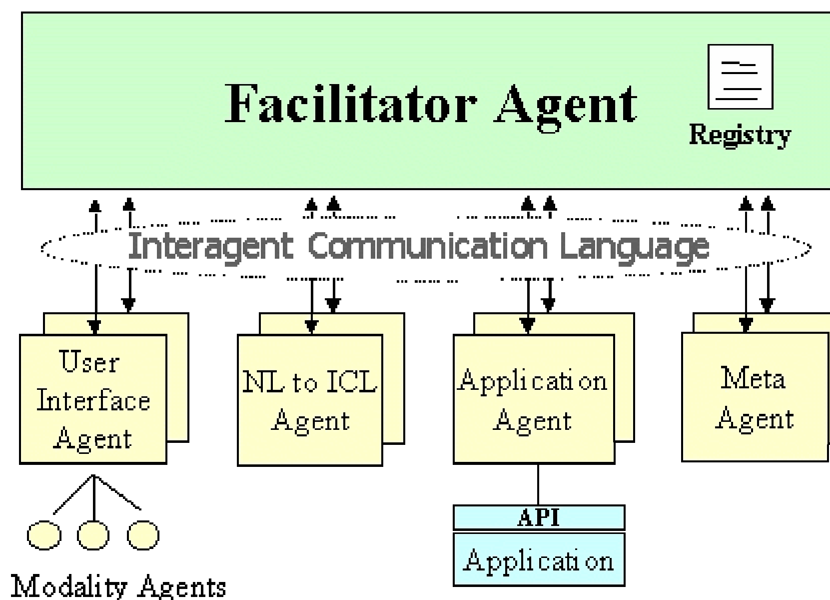


Figure 2: Example OAA agent community instance for a dialog system.

1.3 Pool Architecture (PA)

The **Pool Architecture (PA)** [3] is a multi-blackboard system architecture for distributed systems that consist of independent cooperating modules, which may be running on separate computers under different operating systems. In the VerbMobil project, DFKI developed and continuously improved and extended the Pool Architecture for the multimodal SmartKom system. There are module APIs for the following programming languages: C, C++, Java and PROLOG. The data delivery is based on PVM (Parallel Virtual Machine, [2]). The architecture itself runs under UNIX, LINUX and Windows (NT and 2000).

Modules are realized as independent processes that interact by publishing and subscribing data to and from global data stores (so-called pools). This makes arbitrary communication paths in a dynamic community of distributed processes possible because data producers and data consumers are decoupled. The architecture does not only consist of the necessary infrastructure for inter-process communications. It also contains several kinds of helpful tools for integrators as well as for module developers.

The following paragraphs describe some essential features and details of the Pool Architecture.

1.3.1 The Blackboard Metaphor

Within the blackboard paradigm, modules are seen as *data producers* or *data consumers* that solely interact by exchanging data over blackboards (the so-called *pools* in the PA terminology). In addition to the pools where actual content data is exchanged between the functional modules, there are control pools where control messages and information about the module states are published (e.g., *stateControl* or *moduleState*, which enable the user to see the modules' states on the system GUI). A module can be both, producer and consumer, even to the same pool. To gain access to a pool, the module has to subscribe to this pool from where it reads its input and where it publishes its output. Subscription means that a module wishes to be informed about a specific subject or when a certain event occurs or just at regular intervals. There is no direct communication between the modules that therefore do not have to know of the existence of other modules and their internal workings are invisible from the outside.

Besides, each module can have several instances (e.g., if there are several competing modules with the same functionality).

1.3.2 The Module Manager

The ModuleManager, which builds a uniform interface (API) to the supported programming languages is a special communication layer on top of the Pool Communication layer to hide the details of a complex pool communication protocol from the module programmers. At the beginning, each module informs the ModuleManager about the functionalities it provides and the corresponding pools it wants to subscribe to or publish on. On this basis, the ModuleManager decides whether these pools should be provided with data or not. Furthermore, it controls (i) the reading from pools by providing event masks to inform the modules when data is published on the pools they have subscribed to; (ii) the publishing of data to pools since a module can write on a pool only if it has registered for this pool. For each supported programming language, the ModuleManager provides handlers for managing the control protocol. Furthermore, the ModuleManager allows for synchronization of time-critical processes by introducing internal module states (*connected, waiting, ready, active, inactive*). For example, during the start process, no module should write on a pool before all modules to be started have logged on. During this phase, the modules that have already finished their initialisation process skip from the *connected* state to the *waiting* state. The ModuleManager also provides functions for prioritisation of messages, e.g., some control messages should be made available on the pool and processed with a higher priority.

Several benefits from the introduction of the ModuleManager are gained:

- by hiding some of the more complex protocol tasks from the programmers;
- by adding control information to messages used for internal purposes;
- by deleting old invalid messages as soon as possible to bring the system faster into a defined coherent state faster;
- by hiding the switching between competing modules from the module developer; and
- by making small protocol changes transparent for the users.

1.3.3 The Testbed: Framework for a Distributed Environment

The multi-blackboard architecture used in VerbMobil and SmartKom turned out to be a great advantage for developing a test environment. The open architecture allows for detailed monitoring features, since any process connected to the communication system is in a position to see all interchanged data. Input of data for test purposes can be managed easily at any time and any point. So the prerequisites are given to provide an environment for the specific needs of programmers and users. First of all, there has to be support for developers to enable integration and testing of single modules in stand-alone mode. For this purpose, a modular system is needed, that allows data to be fed into the system at arbitrary module interfaces (pools). Furthermore, there must be a possibility to allow an accurate look at all data produced by the modules involved.
The capabilities of the system have to be demonstrated to diverse groups such as to specialists or to a non-professional audience. Therefore a graphical user interface, easy to handle even by a non-computer specialist and providing attractive visualizations of the module activities is an indispensable requirement for modern system development.
When putting together modules developed at different locations, it is very probable that some cooperation difficulties will occur. Then, some specific debugging tools are needed to effectively trace the reason for malfunction of the system or specific modules.
It is an important feature to control the different processes (representing the single modules) in a running system. This includes "hot swapping" (if there are several modules of the same functionality), killing or restarting a process that got out of hand and changing module-specific adjustments.

Control mechanisms concerning complex correlations without being assigned to one single module can be made available by the environment. Two examples for this strategy in VerbMobil and SmartKom were the opening and closing of the microphone for speech input and the mapping of verbal user commands to concrete module configuration settings.
The possibility of distributing a system via network enables parallel processing and therefore leads to a considerable speeding up of concurrent tasks. Even a distribution beyond the bounds of development sites should be feasible in modern interprocess communication.
All these features were realized in the VerbMobil and SmartKom testbed, a framework serving as a solid basis for the development of modules and usage of the system. Most of these main concepts are not specific to the projects so that they can be utilized by any modular system.

The following figure 3 shows the testbed-GUI for the SmartKom system. The features mentioned above can be accessed via the menu bar. Each button represents a module. The white lines/data buses indicate intermodular data exchange via pools.

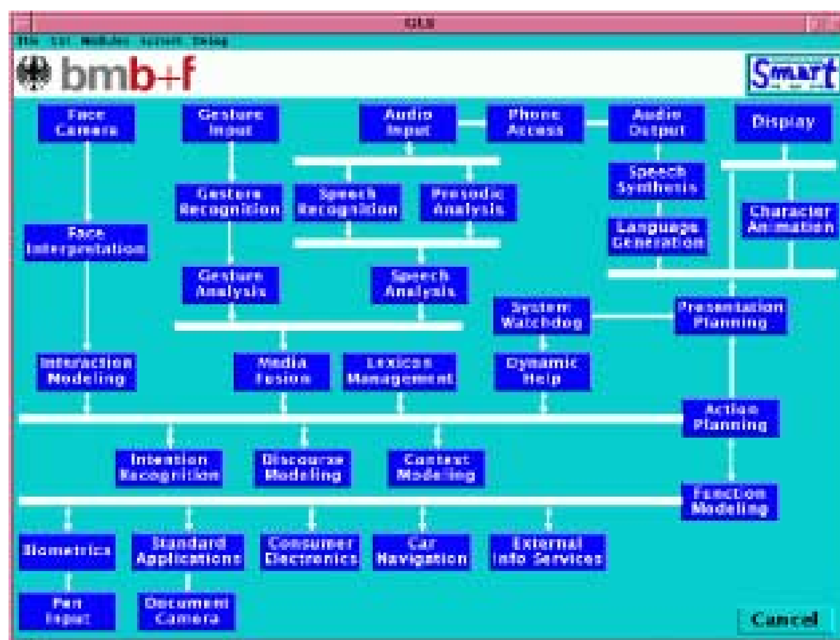


Figure 3: SmartKom-GUI

2 COMIC-specific system architecture comparison

The first task within WP 1 “System Architecture and Integration” was a review of the state of the art by compiling hardware and software modules that were already available at the partners in order to identify project-specific requirements on the architecture imposed by module prerequisites. At the beginning of the project a questionnaire was compiled and distributed to all partners asking for various details concerning their future modules in COMIC in order to collect all architecture relevant requirements coming from partner software prerequisites. Here are some of the most important architecture-relevant items in the partner module questionnaire:

- I/O data (size)
- Programming languages
- Operating systems

Due to the intended system functionalities and hardware definitions of the project several important requirements on COMIC’s system architecture could also be identified from the beginning. The following list combines obligatory system architecture functionalities and features that ease the development, implementation and testing of the COMIC system based on such a system architecture:

- Distributed processing with bi-directional communication
- Multimodal full duplex interaction in mobile environments
- All modules can function in an interrupt-driven real-time environment
- Limited processing power of mobile terminals -> distributed processing
- Minimal effort for adding completely new modules (extendibility)
- Capability of maintaining operation even if some input and/or output channels are unavailable (configurability)

Based on the software module features and the above mentioned COMIC-specific architecture requirements the following criterias were used to compare the three architecture candidates:

- *Message processing/delivery performance*: COMIC will process large data sets (e.g., pen data). Therefore the architecture should have high performance of data delivery for large data sets.
- *Source code availability*: In case of architecture implementation bugs, the source code of the module API’s in all supported programming languages as well as the underlying kernel communication source code should be available for rapid bug fixing.
- *Support*: For system integration as well as for module programmers COMIC needs the best possible support (from the architecture developers as well as from experienced architecture users) to speed up time consuming bug fixes or problems of any kind as far as possible
- *Helpful tools*: The architecture should contain helpful tools that ease integration, running and debugging of the COMIC system. A graphical user interface (GUI) that visualizes important aspects of the overall system and provides easy access to interesting data or interfaces highly eases the system handling and debugging. An elegant start-up tool for distributed runs would allow for a very easy system handling as well. Finally, monitoring and logging tools are very important to ease debugging and testing.

- *Documentation*: The system/architecture handling as well as the module API's in all programming languages and operating systems, the existing tools as well as the system handling should be well documented.
- *Estimated Effort*: Here the estimated effort to adapt the existing architecture to a project specific testbed distribution for COMIC is meant.

The following table gives an overview of the comparison of the three architectures with respect to the above-mentioned criteria:

	Galaxy Communicator	*Open Agent Architecture*	*Pool Architecture*
Performance	High on the same machine under LINUX, unmeasured for distributed runs	Low, there is a inherent 64K size limit to be bypassed	High
Source code availability	For LINUX only (communication software and module API's)	Module API's for LINUX and Windows, but not for the essential facilitator	Module API's and communication software
Support	User mailing list, architecture developer (MITRE Corp.), project partner	User mailing list, architecture developer (SRI), project partner	Best possible, DFKI developer
Tools	No GUI, no start-up tool, for LINUX : debugging, monitoring, data logging	Dynamic GUI, monitor, and debugging tools for LINUX and Windows slowing down the system, start-up uncomfortable	Static GUI, start-up tool, debugging, monitoring and logging (partially LINUX only)
Documentation	Tutorial, module API's for LINUX only	Module API's, system handling and tools	Module API's, system handling and tools (to be translated partially in English)
Effort	Difficult because of missing points for Windows	Feasible if performance problem is solved	Feasible

The table above shows that for each architecture at least one of the essential features is incomplete. It seems that the PA could be the best choice for COMIC because the only really missing point is the completeness of the English documentation while the other two architectures would also need technical extensions with respect to the strong completeness criteria above.

Another difference between the architectures not covered by the table above is the fact that OAA and the Galaxy Communicator are being centrally managed and maintained as open source distributions by MITRE Corp. and SRI, respectively. Thereby the architectures became “stand-alone” architectures that can and have been used in various systems independent of the

modular architecture of the respective systems. On the other hand, PA has not yet reached this open source state. The communication software underlying PA also runs independent of the systems PA is used for. For VerbMobil and SmartKom highly complex individual GUIs had been developed that were individually designed and adapted to the project specific characteristics and needs. While the methods and functionalities that are covered by the GUI in PA are universally usable, they have only been graphically combined with project specific GUI layouts by now. On the other hand, the only task to develop and implement a new project specific GUI is the adaptation of the functional architecture to an appropriate graphical visualisation which is more or less just a design task. While OAA's monitor agent contains a simple hub and spoke layout for graphical visualization of the participated agents within arbitrary agent communities, PA allows to design and implement a project specific GUI layout that especially focuses on and emphasizes project specific features within its layout without loosing any of the mentioned general GUI capabilities.

3 Conclusions & Recommendations

Conclusions:

This document shows that for large and distributed systems the choice of the most appropriate system architecture is a very crucial step to be done in the very beginning. Fully operational architectures that offer high flexibility with respect to communication, coordination, integration, testing and debugging of a large distributed system as well as lowest possible effort for module programmers are to be preferred. Therefore only three architectures were considered in this document.

As shown in the previous section, all three architectures are reasonable candidates for COMIC's system architecture. They all offer easy-to-use module APIs for different programming languages that make the module programming task much easier. There are different kinds of helpful tools for programming modules and debugging systems. For all architectures there are also wide support opportunities by the owner and experienced users as well. The performance of all architectures is also acceptable. Differences lie in the availability of the source code and in the completeness of the distribution for different operating systems.

From COMIC's point of view the PA seems to be the best choice because the architecture has been developed by one of the project partners which ensures the best and quickest possible support within the project. PA would also permit for the fastest possible bug fixing procedure, especially because WP1 "system architecture and integration" is performed by DFKI too. Finally, PA offers the development of a project specific GUI with minor effort. So, from the practicability point of view within COMIC, there is a clear preference on PA.

Recommendations:

Although the comparison above could be improved by more detailed performance measurements for the PA, experiences in VerbMobil and SmartKom have never shown that slow processing of input data is due to the delivery of the data within the architecture but the overall runtime is always an accumulation of the runtimes of the individual modules. Therefore PA is recommended here.

4 References

[1] Galaxy Communicator Web Page, http://communicator.sourceforge.net/, 2002.

[2] A. Geist, A. Bequelin, J. Dongorra, W. Jiang, R. Manchek, and V. Sunderman, „PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing", Cambridge, MA: MIT Press, 1994.

[3] A. Klüter, A. Ndiaye, and H. Kirchmann, "VerbMobil From a Software Engineering Point of View: System Design and Software Integration", in W. Wahlster (ed.), "VerbMobil: Foundations of Speech-to-Speech Translation", ISBN 3-540-067783-6, Springer, 2000.

[4] Labrou, Yannis, and Tim Finin, "A Proposal for a New KQML Specification," Technical Report CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD, February, 1997. Also available online at http://www.cs.umbc.edu/kqml/.

[5] D. L. Martin, A. J. Cheyer, and D. B. Moran, "The Open Agent Architecture: A framework for building distributed software systems," Applied Artificial Intelligence: An International Journal. Volume 13, Number 1-2, January-March 1999. pp 91-128.

[6] Microsoft Corporation, "Distributed Component Object Model Protocol -- DCOM/1.0," January 1998. Available online at http://www.microsoft.com/com/wpaper/default.asp#DCOMpapers.

[7] Object Management Group (OMG), "CORBA/IIOP 2.2 Specification," February 1998. Available online at http://www.omg.org/corba/corbiiop.html.

[8] Sun Microsystems Inc., "Jini(TM) Technology Architectural Overview," Available online at http://www.sun.com/jini/whitepapers/architecture.html.