A Hybrid Architectural Style
for Distributed Parallel Processing of Generic Data Streams

Alexandre R.J. François
University of Southern California
Los Angeles, CA
afrancoi@usc.edu

Abstract

Immersive, interactive applications grouped under the concept of Immersipresence require on-line processing and mixing of multimedia data streams and structures. One critical issue seldom addressed is the integration of different solutions to technical challenges, developed independently in separate fields, into working systems, that operate under hard performance constraints. In order to realize the Immersipresence vision, a consistent, generic approach to system integration is needed, that is adapted to the constraints of research development. This paper introduces SAI, a new software architecture model for designing, analyzing and implementing applications performing distributed, asynchronous parallel processing of generic data streams. SAI provides a universal framework for the distributed implementation of algorithms and their easy integration into complex systems that exhibit desirable software engineering qualities such as efficiency, scalability, extensibility, reusability and interoperability. The SAI architectural style and its properties are described. The use of SAI and of its supporting open source middleware (MFSM) is illustrated with integrated, distributed interactive systems.

1. Introduction

Immersive, interactive applications require on-line processing and mixing of multimedia data such as pre-recorded audio and video, synthetic data generated at run-time, live input data from interaction sensors, media broadcast over a non synchronous channel (e.g. the Internet), etc. The concept of Immersipresence [18] presents many challenges spanning a number of fields, including Signal Processing (compression, 3-D audio), Networking, Database Systems, Computer Vision and Computer Graphics. One challenge seldom addressed is the integration of different solutions developed independently in separate fields into working systems, that must operate under hard optimization constraints (including real-time performance, low latency and precise synchronization). This software engineering issue is often regarded as secondary to the fundamental research aspects in each field. Yet, the actual development of complex cross-disciplinary experiments, from design to implementation, is traditionally limited by actual system integration, which is typically very resource consuming, and source of unforeseen problems.

Traditional industry-oriented, large scale software engineering is not applicable for most research projects. If the ultimate goals are similar (code quality, reusability, efficiency, scalability and interoperability), the constraints and realities of code development in a research environment are simply too different. Research code is developed independently, either by individuals or small teams, whose goal is a proof-of-concept system. In order to realize the Immersipresence vision in the research world, and bring it to the industrial world, a consistent, generic approach to system integration is needed, that is adapted to the constraints of research development. In particular, such a solution must provide significant added value for individuals in order to be adopted, as subsequent integration in cross-disciplinary efforts is not yet a requirement of research development.

This paper introduces SAI, a new software architecture model for designing, analyzing and implementing applications performing distributed, asynchronous parallel processing of generic data streams. SAI aims at providing a universal framework for the distributed implementation of algorithms and their easy integration into complex systems that exhibit desirable software engineering qualities such as efficiency, scalability, extensibility, reusability and interoperability. SAI specifies a new architectural style (components, connectors and constraints). The underlying extensible data model and hybrid (shared repository and message-passing) distributed asynchronous parallel processing model allow natural and efficient manipulation of generic data streams, using existing libraries or native code alike. The modularity of the style facilitates distributed code development, test-

ing, and reuse, as well as fast system design and integration, maintenance and evolution. A graph-based notation for architectural designs allows intuitive system representation at the conceptual and logical levels, while at the same time mapping closely to the physical level.

The paper is organized as follows. section 2 offers a revue of related work in Multimedia and Software Architecture. Section 3 articulates the fundamental design principles underlying the SAI architectural style, formally described in section 4. Section 5 presents application projects that illustrate fundamental architectural patterns, and their combination in the design and implementation of integrated systems in various settings. Section 6 offers concluding remarks.

2. Related work

The issue of (software) system integration is addressed only partially, when at all, in individual fields. Software libraries emerge in specific fields as they reach the necessary levels of maturity. Standard libraries are extremely useful because they provide a common set of data structures and algorithm implementations that can be re-used across applications. As they are designed with a relatively narrow field of application in mind, they do not address interoperability. In particular, such libraries might be built explicitly or implicitly on architectural models that are incompatible. For example, OpenGL [2] and Direct3D [19] both implement a graphics pipeline, with windowing message-based constructs for interaction. Although they can handle animation to some extent (through extensions of the message-based model), they do not explicitly model generic data streams. DirectShow [19] implements a Dataflow architecture for data-stream processing. Although Direct3D and DirectShow are part of the DirectX [19] set of multimedia libraries, they are inherently incompatible, in the sense that they do not share either a common, or compatible architectural models.

2.1. Multimedia

The dramatic development of networks and the Internet made "multimedia" the field of research dealing essentially with *storage, retrieval, transmission and presentation* of large amounts of data. The multi- prefix is justified by the handling of data that might not be text (e.g. images and sound), and that is usually in the form of bandwidth-intensive data *streams*. The goal is to deliver high quality media. Because of the predominance of the communications aspect, system architecture in this context usually refers to the underlying networking topology (e.g. client-server or peer-to-peer). These network-centric architectural concerns however do not address the internal organization of the various components, which is usually where the major cross-disciplinary issues arise. Bandwidth being regarded as the major limiting factor, the only on-line (real-time) data processing performed is related to compression and decompression. Presentation does not involve much processing beyond decompression and display, which justifies the adoption of Pipes and Filters variants, dataflow-based approaches in DirectShow and all other so called "media streaming" libraries and packages, in research and industry alike. Example research projects include MIT's VuSystem [15], the Berkeley Continuous Media Toolkit [17], the Network Integrated Media Middleware [16], and the Distributed Media Journaling (DMJ) project [6]. These efforts all rely on modular dataflow architecture concepts. They are designed primarily for audio and video on-line processing and transmission, with a strong emphasis on capture, transmission and replay aspects. They cannot easily scale up to applications involving immersion, interaction and synthetic content mixing.

Multimedia storage and retrieval are traditionally database systems issues. The main challenge here lies in the amount of data manipulated. From a storage point of view, the limiting factor is again bandwidth. Beyond physical access and transmission constraints, retrieval raises more than ever the issue of content analysis and understanding. Pure signal processing techniques for content analysis (e.g. of music, images and video) have very limited success, while Artificial Intelligence related techniques remain confined to limited domains of application. Thus human intervention, through annotation tools, remains the most reliable and useful method. Similarly, content development is almost exclusively a human task, addressed through interactive tools, distinct from multimedia systems. In such systems, consumers are fed views (in the database sense), and have very limited control over the data that is presented to them: classical database queries for retrieval, VCR-style commands for streams. Personalization of the content provided is marginal, little or no content is synthesized on the fly.

Incidentally, the architectural model of choice for multimedia streaming systems (Dataflow) is not appropriate for interactive media content creation tools, which all run in a windowing message-based interactive environment. The model underlying interaction tools is usually data-driven: a video stream for example is considered as an object whose components (frames) can be edited in a random access fashion, not as a sequential stream of data. This is for example the basis for the "pool of frames" approach to temporal media data handling in the MVC architecture [14]. Similarly, Artificial Intelligence approaches deal with dynamic but persistent data structures, as opposed to Signal Processing techniques which deal with data streams.

2.2. Games

Distributed, first person collaborative games have probably more in common with the Immersipresence vision than any other type of multimedia systems. This stems from the fact that games are first and foremost highly interactive applications. The networking aspect has only been introduced recently, after great emphasis was given to sound and graphics quality, and simultaneously with other advanced facets of game development such as realistic physics simulation and use of Artificial Intelligence. As a result, one might expect game engines to take after architectural designs allowing to consistently integrate various technologies and thus directly applicable for Immersipresence.

Where video-on-demand and other high-fidelity media streaming applications are yet to become business realities, gaming is actually an already profitable, and fast growing industry (US$6.9 billion in US sales in 2002 according to [1]), subject to market imposed constraints. The most advanced designs (including architectural designs) are proprietary, and constitute valuable intellectual property for game development companies. Game development projects have become multi-year team productions, with multi-million dollar budgets [13], and industrial-size management problems. More importantly, game engines are by design extremely fine-tuned systems with very specific hardware constraints, that do not necessarily scale-up for architectural generalization. Furthermore, time constraints in new developments usually translate into quick fixes rather than redesign of existing software. Various commercial so-called middleware packages are available, that provide support for specific aspects such as physics simulation, artificial intelligence, networking, etc. Their use in particular projects is determined by the amount of work needed to interface them with a given game engine [5, 7], which often outweighs the advantages of reuse. This reinforces the view that game engines are not built as modular, extensible platforms, but rather fine-tuned, ad-hoc systems designed to get the most out of a specific target hardware configuration. On the academic side, game design is only very slowly making its way as a valid engineering research topic, and thus the development and study of generic architectures is not a mainstream effort.

2.3. Software architecture

It is interesting to notice that the set of issues raised in the development of modern games is largely disjoint from that of issues raised by the development of multimedia streaming systems. Although core technologies might find applications in both fields, they are two totally distinct efforts at the system architecture level. Yet, both efforts can be unified in the quest for Immersipresence, which requires both a high level of interaction and a high fidelity experience.

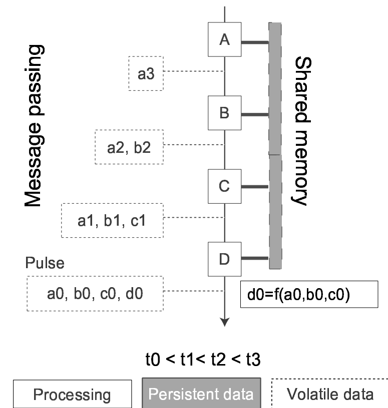

Figure 1. A hybrid message passing and shared-repository model for distributed asynchronous parallel processing of data streams.

The Pipes and Filters (or other dataflow variants) architectural style favored in “streaming” applications, has a number of well known positive properties [20] that make it an attractive candidate for real-time distributed parallel processing of data streams. As a modular model, it is relatively simple and intuitive. Localization and isolation of computations facilitates system design, implementation, maintenance and evolution. Because filters are independent, the model naturally supports parallel and distributed processing. However, a few key shortcomings and limitations make it unsuitable for designing cross-disciplinary dynamic systems, possibly involving real-time constraints, user immersion and interaction [10]. Major areas of limitation are efficiency and modeling power. In particular, the absence of a clear mechanism for shared data access accounts for the inability of the Pipes and Filters style to apply to data driven (including symbolic computations) problems. On the other hand, data-driven architectures such as Blackboards and other shared repository models, are not adapted to real-time on-line processing of data streams. Architectures with higher level of abstraction, or simple juxtaposition of similar low level architectural styles do not resolve this incompatibility: a new, hybrid style is needed.

3. Design principles

A few key observations, resulting from the analysis of system requirements for real-time interactive, immersive applications, allow to formulate principles and concepts that address the shortcomings identified above.

3.1. Time

A critical underlying concept in all user-related application domains (but by no means limited to these domain) is that of time. Whether implicitly or explicitly modeled, time relations and ordering are inherent properties of any sensory-related data stream (e.g. image streams, sound, haptics, etc.), absolutely necessary when users are involved in the system, even if not on-line or in real-time. Users perceive data as streams of dynamic information, i.e. evolving in time. This information only makes sense if synchronization constraints are respected within each stream (temporal precedence) and across streams (precedence and simultaneity). It follows that time information is a fundamental attribute of all process data, and should therefore be explicitly modeled both in data structures and in processes.

Synchronization is a fundamental operation in temporal data stream manipulation systems. It should therefore also be an explicit element of the architectural model. A structure called *pulse* is introduced to regroup synchronous data (see figure 1). Data streams are thus quantized temporally (not necessarily uniformly). As opposed to the Pipes and Filters case, where data remains localized in the filters where it is created or used, data is now grouped in pulses, which flow from processing center to processing center along streams. The processing centers do not consume their input, but merely use it to produce some output that is added to the pulse. This also reduces the amount of costly data copy: in a subgraph implemented on a platform with shared memory space, only a pointer to the evolving pulse structure will be transmitted from processing center to processing center. Note that such processing centers can no longer be called filters.

3.2. Parallelism

Figure 2 illustrates system latency, which is the overall computation time for an input sample, and throughput or output rate, inverse of the time elapsed between two consecutive output samples. The goal for high quality interaction is to minimize system latency and maximize system throughput. In the sequential execution model, latency and throughput are directly proportional. In powerful computers, this usually results in the latency dictating the system throughput as well, which is arguably the worst possible case. In the Pipes and Filters model, filters can run in parallel. Latency and throughput are thus independent. Because of the parallelism, system latency can be reduced in most cases with careful design, while system throughput will almost always be greatly improved. The sequential behavior of the pipes, however, imposes on the whole system the throughput of the slowest filter. This constraint can actually be relaxed to yield an asynchronous parallel processing model.

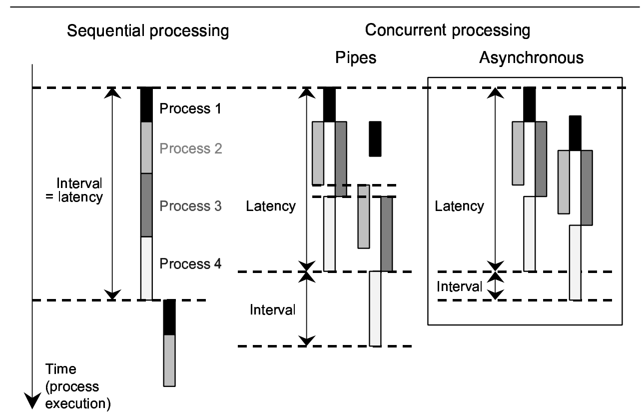

Figure 2. Advantage of parallelism for time sensitive applications. Processes 2 and 3 are independent and both depend on process 1; Process 4 depends on both 2 and 3.

Instead of being held in a buffer to be processed by order of arrival, each incoming pulse is processed on arrival in the processing center, in a separate thread. *Achievable* throughput is now optimal. It will actually be achieved if no hardware or software resources become exhausted (e.g. computing power, memory, bus bandwidth, etc.). Of course, an asynchronous model requires to explicitly implement synchronization when necessary, but *only then*.

3.3. Data classes

The Pipes and Filters model explicitly separates data streams and process parameters, which is both a valid functional distinction, and a source of inconsistency in the model, leading to important limitations as explained above. A re-consideration of this categorization, in the context of temporal data streams processing, reveals two distinct data classes: *volatile* and *persistent*.

Volatile data is used, produced and/or consumed, and remains in the system only for a limited fraction of its lifetime. For example, in a video processing application, the video frames captured and processed are typically volatile data: after they have been processed and displayed or saved, they are not kept in the system. Process parameters, on the other hand, must remain in the system for the whole duration of its activity. Note that their value can change in time. They are dynamic yet persistent data.

All data, volatile or persistent, should be encapsulated in pulses. Pulses holding volatile data flow down streams defined by connections between the processing centers, in a message passing fashion. They trigger computations, and are thus called *active* pulses. In contrast, pulses holding persistent information are held in repositories, where the

processing centers can access them in a concurrent shared memory access fashion. This hybrid model combining message passing and shared repository communication (outlined in figure 1), combined with a unified data model, provides a universal processing framework. In particular, feedback loops can now be explicitly and consistently modeled.

From the few principles and concepts outlined above emerged a new architectural style. Because of the context of its development, the new style was baptized SAI, for "Software Architecture for Immersipresence."

4. The SAI Style

This section offers a formal definition of the SAI architectural style. Graphical symbols are introduced to represent each element type. Together these symbols constitute a graph-based notation system for representing architectural designs. In addition, when available, the following color coding will be used: green for processing, red for persistent data, blue for volatile data. Figure 3 presents a summary of the proposed notation. In the remainder of this paper, the distinction between an object type and an instance of the type will be made explicitly only when required by the context.

4.1. Components, connectors and constraints

The SAI style defines two types of components: *cells* and *sources*. Cells are processing centers. They do not store any state data related to their computations. The cells constitute an extensible set of specialized components that implement specific algorithms. Each specialized cell type is identified by a type name (string), and is logically defined by its input data, its parameters and its output. Cell instances are represented graphically as green squares. A cell can be active or inactive, in which case it is transparent to the system. Sources are shared repository of persistent data. Source instances are represented as red disks or circles. Two types of connectors link cells to cells and cells to sources. Cell to source connectors give the cell access to the source data. Cell to cell connectors define data conduits for the streams. The semantics of these connectors are relaxed compared to that of pipes (which are FIFO queues): they do not convey any constraint on the time ordering of the data flowing through them.

Cell and source instances interact according to the following rules. A cell must be connected to exactly one source, which holds its persistent state data. A source can be connected to an arbitrary number of cells, all of which have concurrent shared memory access to the data held by the source. A source may hold data relevant to one or more of its connected cells, and should hold all the relevant data for each of its connected cells (possibly with some overlap). Cell-source connectors are drawn as either double or fat red lines. They may be drawn across cells (as if cells were connected together by these links) for layout convenience. Volatile data flows in *streams*, which are defined by cell-to-cell connections. A cell can be connected to exactly one upstream cell, and to an arbitrary number of downstream cells. Streams (and thus cell-cell connections) are drawn as thin blue arrows crossing over the cells.

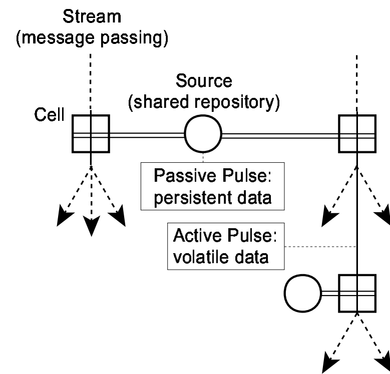

Figure 3. Summary of SAI notation.

4.2. Data model

Data, whether persistent or volatile, is held in *pulses*. A pulse is a carrier for all the synchronous data corresponding to a given time stamp in a stream. Information in a pulse is organized as a mono-rooted composition hierarchy of *node* objects. The nodes constitute an extensible set of atomic data units that implement or encapsulate specific data structures. Each specialized node type is identified by a type name (string). Node instances are identified by a name. The notation adopted to represent node instances and hierarchies of node instances makes use of nested parentheses, e.g.: (NODE_TYPE_ID "Node name" (...) ...). This notation may be used to specify a cell's output, and for logical specification of active and passive pulses.

Each source contains a *passive pulse*, which encodes the instantaneous state of the data structures held by the source. Volatile data flows in streams, that are temporally quantized into *active pulses*.

4.3. Processing model

When an active pulse reaches a cell, it triggers a series of operations that can lead to its processing by the cell (hence the "active" qualifier). Processing in a cell may result in the augmentation of the active pulse (input data), and/or update of the passive pulse (process parameters). The processing of active pulses is carried in parallel, as they are re-

ceived by the cell. Since a cell process can only read the existing data in an active pulse, and never modify it (except for adding new nodes), concurrent read access will not require any special precautions. In the case of passive pulses, however, appropriate locking (e.g. through critical sections) must be implemented to avoid inconsistencies in concurrent shared memory read/write access.

4.4. Dynamic data binding

Passive pulses may hold persistent data relevant to several cells. Therefore, before a cell can be activated, the passive pulse must be searched for the relevant persistent data. As data is accumulated in active pulses flowing down the streams through cells, it is also necessary for a cell to search each active pulse for its input data. If the data is not found, or if the cell is not active, the pulse is transmitted, as is, to the connected downstream cells. If the input data is found, then the cell process is triggered. When the processing is complete, then the pulse, which now also contains the output data, is passed downstream.

Searching a pulse for relevant data, called *filtering*, is an example of run-time data binding. The target data is characterized by its structure: node instances types and names and their relationships. The structure is specified as a *filter* or a composition hierarchy of filters. Note that the term filter is used here in its “sieving” sense. Figure 4 illustrates this concept. A filter is an object that specifies a node type, a node name or name pattern and eventual subfilters corresponding to subnodes. The filter composition hierarchy is isomorphic to its target node structure. The filtering operation takes as input a pulse and a filter, and, when successful, returns a *handle* or hierarchy of handles isomorphic to the filter structure. Each handle is essentially a pointer to the node instance target of the corresponding filter. When relevant, optional names inherited from the filters allow to identify individual handles with respect to their original filters.

The notation adopted for specifying filters and hierarchies of filters is nested square brackets. Each filter specifies a node type, a node instance name or name pattern (with wildcard characters), an optional handle name, and an eventual list of subfilters, e.g.: [NODE_TYPE_ID “Node name” *handle_id* [...] ...]. Optional filters are indicated by a star, e.g.: [NODE_TYPE_ID “Node name” *handle_id*]*.

When several targets in a pulse match a filter name pattern, all corresponding handles are created. This allows to design processes whose input (parameters or stream data) number is not fixed. If the root of the active filter specifies a pattern, the process method is invoked for each handle generated by the filtering (sequentially, in the same thread). If the root of the passive filter specifies a pattern, only one passive handle is generated (pointing to the first encountered node satisfying the pattern).

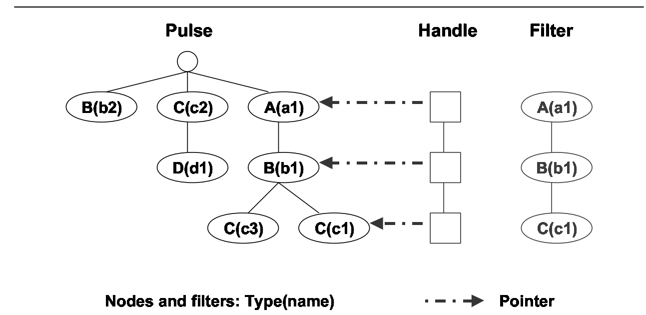

Figure 4. Data binding: pulse filtering.

4.5. Architectural design specification

A particular system architecture is specified at the conceptual level by a set of source and cell instances, and their inter-connections. The specialized cells may be accompanied by a description of the task they implement. Source and cell instances may be given names for easy reference. In some cases, important data nodes and outputs may be specified schematically to emphasize some design aspects. Section 5 shows such a conceptual graphs for an example integrated system architecture.

A logical level description of a design requires to specify, for each cell, its active and passive filters and its output structure, and for each source, the structure of its passive pulse. Table 1 summarizes the notations for logical level cell definition. Filters and nodes are described using the nested square brackets and nested parentheses notations introduced above. By convention, in the cell output specification, (x) represents the pulse’s root, (.) represents the node corresponding to the root of the active filter, and (..) represents its parent node.

4.6. Architectural Middleware

MFSM (Modular Flow Scheduling Middleware) [8] is an open source architectural middleware implementing the core elements of the SAI style. MFSM aims at promoting and supporting the design, analysis and implementation of applications in the SAI style. A number of software modules regroup specializations implementing specific algorithms or functionalities. They constitute a constantly growing base of open source, reusable code, maintained as part of the MFSM project. The project also comprises extensive documentation, including user guide, reference guide and tutorials.

4.7. Architectural properties

By design, the SAI style shares many of the desirable properties identified in the Pipes and Filters model. It allows

ClassName (ParentClass)	CELL_TYPE_ID
Active filter	[NODE_TYPE_ID "Node name" *handle_id* [...] ...]
Passive filter	[NODE_TYPE_ID "Node name" *handle_id* [...] ...]
Output	(NODE_TYPE_ID "*default output base name* –more if needed" (...) ...)

Table 1. Notations for logical cell definition.

intuitive design, emphasizing the flow of data in the system. The graphical notation for conceptual level representations give a high level picture that can be refined as needed, down to implementation level, while remaining consistent throughout. The high modularity of the model allows distributed development and testing of particular elements, and easy maintenance and evolution of existing systems. The model also naturally supports distributed and parallel processing. Unlike the Pipes and Filters style, the SAI style provides unified data and processing models for generic data streams. It supports optimal (theoretical) system latency and throughput thanks to an asynchronous parallel processing model. It provides a framework for consistent representation and efficient implementation of key processing patterns such as feed-back and interaction loops and incremental processing along the time dimension (see section 5).

The SAI style has several other important architectural properties, including natural support for dynamic system evolution, run-time reconfigurability and self monitoring. Although these are not fully utilized in current application projects, they are actively investigated and will be a the core of upcoming projects.

Finally, a critical architectural property that must be considered is *performance overhead*. Some aspects of the SAI data and processing models, such as filtering, involve non trivial computations, and could make the theory impractical. The existence of fairly complex systems designed in the SAI style and implemented with MFSM show that, at least for these examples, it is not the case. Experimental scalability tests performed on applications designed in the SAI style, reported in [12] and [10], suggest that: (1) as long as computing resources are available, the overhead introduced by the SAI processing model remains constant, and (2) the contribution of the different processes are combined linearly. In particular, the model does not introduce any nonlinear complexity in the system. These properties are corroborated by empirical results and experience in developing and operating other systems designed in the SAI style. Theoretical complexity analysis and overhead bounding are ongoing research efforts.

5. Example designs

SAI and MFSM have been used for the design and implementation of various experimental systems. A number of projects ranging from single stream automatic real-time video processing to fully integrated distributed interactive systems mixing live video and graphics are presented in [10]. Specific projects include real-time segmentation and tracking [12], a handheld mirror simulation [11], a scene-graph based graphics toolkit [9]. This sections offers overviews and discussions of three recent examples involving video, graphics, sound and other multimedia data streams and structures.

5.1. MuSA.RT

MuSA.RT (Music on the Spiral Array . Real-Time), Opus 1 [4], is a system for real-time analysis and interactive visualization of tonal patterns in music.

MIDI input, for example from a live performance, is processed, analyzed and mapped in real-time to the Spiral Array [3], a 3D model for tonality. Contextual tonal information is summarized in a center of effect (CE), which maps any pitch collection to a spatial point and any time series of notes to meaningful trajectories inside the Spiral Array. CE trajectory analysis allows to infer the presently active set of

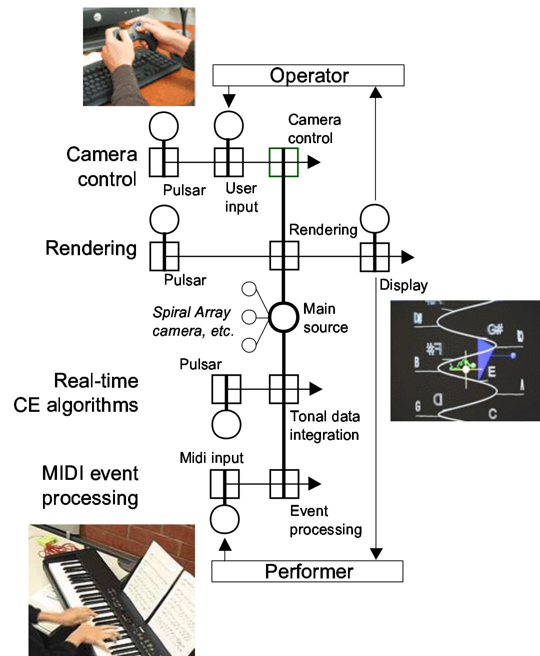

Figure 5. MuSA.RT system graph.

pitch classes, and higher level constructs, such as the current chord and key, revealed to the spectators through real-time 3D rendering. An operator can concurrently navigate through the Spiral Array space using a gamepad to zoom in and out, tilt the viewing angle and circle around the spiral to get a better view of the tonal structures. An automatic pilot option seeks the best view angle and centers the camera at the heart of the action. Figure 5 shows the system conceptual graph.

The system consists of four independent data streams, each exhibiting general architectural patterns: (1) MIDI input and event processing; (2) tonal analysis (real-time CE algorithms); (3) rendering of the Spiral Array structures; and, (4) control device (gamepad) input and camera manipulation. These four streams potentially operate according to different modalities (e.g. push or pull input models) and at different rates. The spiral array structure, processing and rendering parameters are persistent (yet dynamic) data; the MIDI messages and the rendered frames for visualization are volatile data. The application graph contains two interaction loops, one involving the performer, the other involving the pilot.

One major difficulty in building systems such as MuSA.RT is the coexistence and synchronization of multiple data streams processed and synthesized in real-time. Furthermore, the complexity of such cross-disciplinary experiments is traditionally limited by actual system integration, which is the main source of unforeseen problems. Using SAI and MFSM greatly simplified system design, implementation and integration. The Opus 1 system constitutes a platform for testing and validating the different modules involved. Each functional module can be replaced by a functionally equivalent module, allowing to conduct strictly controlled comparisons in an otherwise identical setting.

5.2. IMSC Communicator

The IMSC Communicator is an experimental extensible platform for remote collaborative data sharing. The architectural patterns highlighted here directly apply to the design of any type of distributed applications.

Popular architectures for communication applications include Client/Server and Peer To Peer. Different elements of the overall system are considered separate applications. Although these models can either be encapsulated or implemented in the SAI style, a communication application designed in the SAI style can also be considered a single distributed application graph, in which some cell to cell connections are replaced with network links. From this point of view, specific network architectures would be implemented in the SAI style, as part of the overall distributed application.

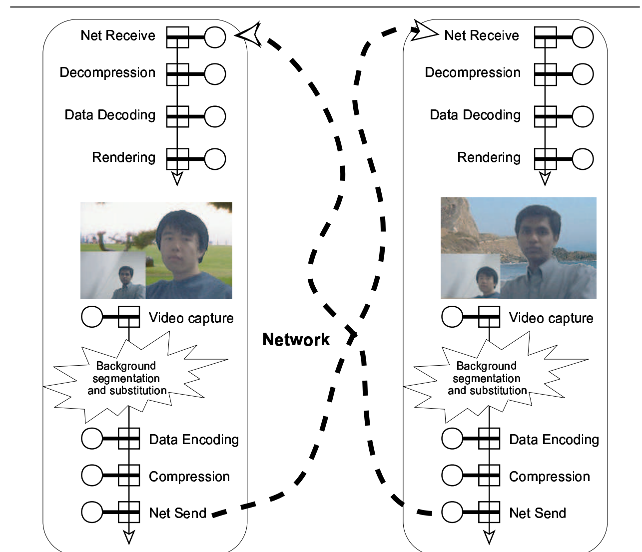

Figure 6. IMSC Communicator flow graph.

The core pattern of the Communicator is a sequence of cells introducing network communication between two independent subgraphs.

Figure 6 shows the conceptual graph for a 2-way communicator supporting image and sound, with example screen shots. The communication pattern comprises cells for encoding, compression and networking (send) on the emitting side, networking (receive), decompression and decoding on the receiving side. The encoding cell flattens a node structure into a linear buffer so that the structure can later be regenerated. The compression cell takes as input the encoded character string produces a compressed buffer. Note that the compression step is optional. The networking cells are responsible for packetizing and sending incoming character strings on one side, and receiving the packets and restoring the string on the other side. The decoding cells regenerate the node structure into a pulse, from the encoded character string.

Once a generic platform is available for developing and testing data transfer modalities, support for various specific data types can be added. Different modalities and protocols can be implemented and tested. In the particular system presented here, a background replacement unit, based on the segmentation by change detection described in [12], was added to the capture side to illustrate how the modularity of the architecture allows to "plug-and-play" subgraphs developed independently. Support for, and parallel independent processing of image and sound data demonstrates the advantages of adopting an asynchronous model, and of performing synchronization only when necessary, in this case in the rendering cell.

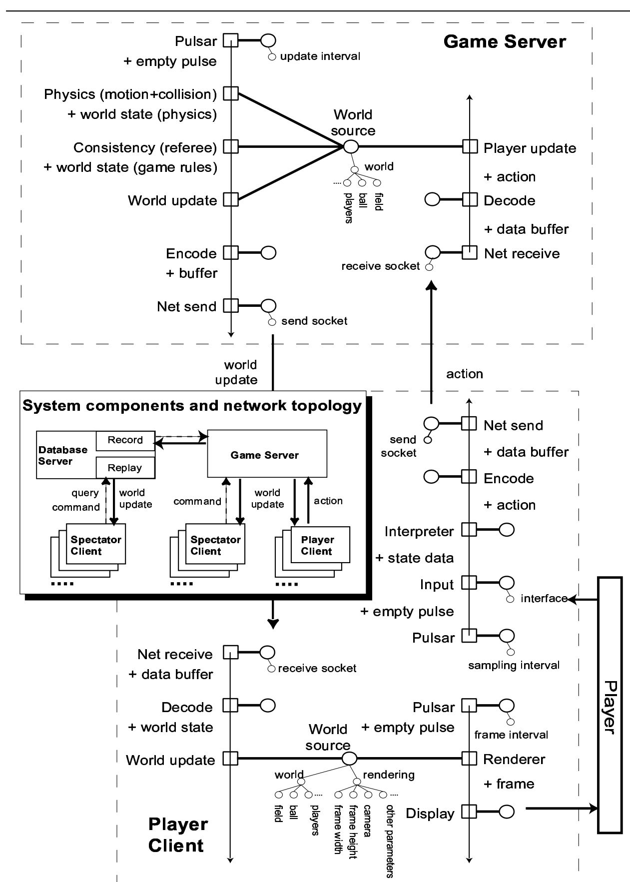

Figure 7. A distributed soccer game.

5.3. Distributed soccer game

In the fall of 2002, the USC Computer Science department offered the first course specifically dedicated to media systems integration. This graduate level seminar course, entitled "Integrated Media Systems," introduced students to the specific technological and organizational difficulties of designing and building interactive, media rich integrated systems. SAI was central to the design and implementation of the course. Multimedia processing and application integration techniques were illustrated in a distributed class project. Using MFSM, the 25 students, developed a playable on-line (simplified) soccer game in just two months. None of the students had used SAI before, and none had ever participated in such a project.

Figure 7 presents a conceptual graph of game components (network topology) and example SAI graphs for game server and player client. Small teams (1-8 students) developed independent modules implementing key functionalities in different areas of media processing: networking (client/server), database (game recording and replay), rendering, physics/gameplay, interaction devices (including gamepad-based control and cyberglove-based gesture recognition control). Module development time was kept to a minimum by encapsulating existing libraries whenever possible (e.g. OpenGL [2] for graphics, DirectPlay [19] for rendering, etc.). The modules were then used to build the different elements forming the game system (game and database servers, player and spectator clients). Using SAI and MFSM allowed to create a modular design, and to establish and follow a rigorous module development and cross-testing schedule, so that the final integration was only a matter of compiling the different executables composing the complete system. Figure 8 shows students performing extensive testing of the project.

This course represents a major experiment in the teaching of "multimedia," by departing from the traditional "list of topics" approach. It spectacularly demonstrates the efficiency of SAI as a framework for distributed development of integrated systems. It also validates MFSM not only as an invaluable design and development tool, but also as a learning tool, that the students can take from the classroom to the research or development environment. Encouraged by the enthusiastic student reviews and feedback received for the class, regular graduate and undergraduate level courses applying this type of project approach are in the planning.

6. Summary and perspectives

This paper introduced SAI, a new software architecture model for designing, analyzing and implementing applications performing distributed, asynchronous parallel processing of generic data streams.

SAI specifies a new architectural style (components, connectors and constraints). The underlying extensible data model and hybrid (shared repository and message-passing) distributed asynchronous parallel processing model allow natural and efficient manipulation of generic data streams, using existing libraries or native code alike. The modularity of the style facilitates distributed code development, testing, and reuse, as well as fast system design and integration, maintenance and evolution. A graph-based notation for architectural designs allows intuitive system representation at the conceptual and logical levels, while at the same time

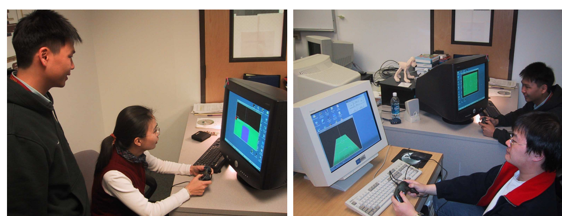

Figure 8. Students testing the project.

mapping closely to processes. Architectural patterns were illustrated with several example integrated systems. By design, the SAI style preserves desirable properties identified in some classic architectural styles. Beyond these, SAI allows to achieve optimal (theoretical) system latency and throughput, and provides a unified framework for consistent representation and efficient implementation of fundamental processing patterns such as feed-back loops and incremental processing along the time dimension. SAI is supported by MFSM, an open source architectural middleware.

The SAI architectural style exhibit properties that make it relevant to research, educational and industrial projects. SAI is well suited to distributed development of functional modules, and their seamless integration into complex systems. The modularity of the design allows gradual development, facilitating continuing validation and naturally supporting regular delivery of incremental system prototypes. A number of cross-disciplinary research projects are already leveraging these properties. They are producing real-time, interactive systems spanning a range of research domains. Using the SAI style in research projects may also facilitate technology transfer to industry. In industry, the SAI style allows fast prototyping for proof-of-concept demonstrations, and may prove to be a valuable framework to complement modern software engineering practices. For education, SAI allows to efficiently and realistically relate classroom projects to the realities of the research laboratory, and of industrial software development, as demonstrated by the Fall 2002 Integrated Media Systems course experiment at USC.

Applications of SAI in various contexts, especially in the design and integration of real-time interactive systems, are currently being explored. One example is an integrated design and development environment for the SAI style. On the theoretical side, the relations between SAI and existing architecture description languages and related architectural styles are of prime interest, as their study and understanding should result in theoretical and practical tools for architecture design validation and analysis.

7. Acknowledgments

This work has been funded in part by the Integrated Media Systems Center, a National Science Foundation Engineering Research Center, Cooperative Agreement No. EEC-9529152. Any Opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect those of the National Science Foundation.

References

[1] Entertainment software association. *www.theesa.com*.

[2] Opengl. *www.OpenGL.org*.

[3] E. Chew. *Towards a Mathematical Model of Tonality*. Ph.d. thesis, MIT, Cambridge, MA, 2000.

[4] E. Chew and A. R. J. François. MuSA.RT - Music on the Spiral Array . Real-Time. In *Proc. ACM Multimedia*, Berkeley, CA, USA, Nov. 2003.

[5] E. Dybsand. AI middleware: Getting into character (parts 1-5). *www.gamasutra.com*, July 2003.

[6] V. Eide, F. Eliassen, O.-C. Granmo, and L. O. Scalable independent multi-level distribution in multimedia content analysis. In *Proc. Joint Int. Wks. Interactive Distributed Multimedia Systems / Protocols for Multimedia Systems (IDMS/PROMS 2002)*, Coimbra, Portugal, 2002.

[7] M. Ferguson and M. Ballbach. Product review: Massively multiplayer online game middleware. *www.gamasutra.com*, Jan. 2003.

[8] A. R. François. Modular Flow Scheduling Middleware. *mfsm.SourceForge.net*.

[9] A. R. François. Components for immersion. In *Proc. IEEE Int. Conf. Multimedia and Expo*, Lausanne, Switzerland, Aug. 2002.

[10] A. R. François. Software architecture for computer vision: Beyond pipes and filters. Technical Report IRIS-03-240, Institute for Robotics and Intelligent Systems, USC, 2003.

[11] A. R. François and E. Kang. A handheld mirror simulation. In *Proc. IEEE Int. Conf. Multimedia and Expo*, Baltimore, MD, July 2003.

[12] A. R. François and G. G. Medioni. A modular software architecture for real-time video processing. In *IEEE Int. Wks. Computer Vision Systems*, pages 35–49, Vancouver, B.C., Canada, July 2001.

[13] Gamasutra.com. Postmortem feature article series. *www.gamasutra.com*.

[14] N. T. Graham and T. Urnes. Scalable independent multi-level distribution in multimedia content analysis. In *Proc. Joint Int. Wks. Interactive Distributed Multimedia Systems / Protocols for Multimedia Systems (IDMS/PROMS 2002)*, Coimbra, Portugal, 2002.

[15] C. Lindblad and D. Tennenhouse. The VuSystem: A programming system for compute-intensive multimedia. *IEEE Jour. Selected Areas in Communications*, 14(7):1298–1313, Sept. 1996.

[16] M. Lohse, M. Repplinger, and P. Slusallek. An open middleware architecture for network-integrated multimedia. In *Proc. Joint Int. Wks. Interactive Distributed Multimedia Systems / Protocols for Multimedia Systems (IDMS/PROMS 2002)*, Coimbra, Portugal, 2002.

[17] K. Mayer-Patel and L. Rowe. Design and performance of the Berkeley Continuous Media Toolkit. In M. Freeman, P. Jardetzky, and H. Vin, editors, *Multimedia Computing and Networking*, pages 194–206. 1997.

[18] D. McLeod, U. Neumann, C. Nikias, and A. Sawchuk. Integrated media systems. *IEEE Signal Processing Mag.*, 16(1):33–43, Jan. 1999.

[19] Microsoft. Directx. *www.microsoft.com/directx*.

[20] M. Shaw and D. Garlan. *Software Architecture - Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, NJ, 1996.