Sportlink: a Sports Management Resource Planning Ecosystem

Arjen Schoneveld and Frank Lyaruu

Dexels BV, Asterweg 20D2, 1031HN, Amsterdam, the Netherlands
aschoneveld@dexels.com, flyaruu@dexels.com
WWW home page: http://www.dexels.com/

Abstract. In this paper we discuss an architecture for, a large scale, *Software as as Service* (SaaS) based application: a *Sports Management Resource Planning* system. Many SaaS applications follow a monolithic, three tier, application stack using a browser based front end. The incompatibilities between customers, in our case sports assocations, and the diversity of end-user roles are not very well captured by such a monolithic architecture. Also, considering software durability and scarcity of financial resources within sports associations, the application must have a very long lifetime expectancy. Our challenge is to design an application that adheres to all of these requirements and that is stable in a possibly hostile environment during 24x7 operation. In this paper we suggest a Service Oriented Computing approach augmented with a potentially unlimited amount of front end applications to support association incompatabilities and the different end-user roles.

Key words: agile software architecture, software as a service, monitoring agents, software development approach, sports information systems

1 Introduction

In this paper we will present the architecture of a specialized Enterprise Resource Planning (ERP) system, a Sports Management Resource Planning System (SMRP) called *Sportlink*[1]. First, we will describe the environment of the system. Sportlink is an enterprise information system used by a large number of sports associations in the Netherlands. The Sportlink system offers many functional modules that can be used in various sports organization related processes. These functional modules have been set up as generic as possible in order to offer a single *SaaS* solution for many different sports associations. Different sports associations use different business rules that also require different third party application integration. Within a sport association Sportlink serves a variety of end-users that interact and cooperate in several business processes. The system is used by a large number of sports organizations at different organizational levels, volunteers, athletes and various other stakeholders with different interests. Each of these stakeholders is involved in parts of the organizational process. The

[1] http://www.sportlink.com/

association is in charge of coordinating different tasks, while sub tasks are divided among other entities at lower organizational levels. Each of these different entities have different requirements, they need different views and corresponding functionality to perform their part of the organizational process. Furthermore, all these different stakeholders change and co-evolve, each at their own pace.

To implement such a system, a widely used architectural approach would be that of a typical web application following a three-tier layered model: a web based front-end, an application server and a back-end database. It would be very hard to configure such an architecture for the wide range of customer functional- and business-rule requirements. We could of course decide to implement seperate application instances for each customer, however, this would surely lead to a maintainability nightmare. We have chosen a Service Oriented Computing (SOC)[4] approach instead. The main ingredients of our architecture are: a plethora of front-end applications, a service layer and a monitoring agent-based infrastructure. Many of the different front-end applications are third party systems that require services from the Sportlink system. Other front-end applications are considered to be part of the core Sportlink solution and include a sports association application, a club application, an official portal and a mobile digital match form application. For optimal alignment with the service layer we introduce a declarative user interface framework: *Tipi*. A flexible service run-time, *Navajo*, is used to host the many services that compose the Sportlink system. The feasibility of our approach is demonstrated by showing the successful implementation of the Sportlink system by a large number of sports associations in the Netherlands.

This paper is structured as follows. In the next section we will sketch typical characteristics of the Sports Enterprise and derive important non-functional requirements (functional requirements are without the scope of this paper). In the third section we discuss architectural styles as the ingredients for our solution. In the fourth section, a software architecture will be introduced that is designed to match the non-functional requirements. A feasibility study, in section five, describes the behavior of Sportlink in the real world, considering performance- and adaptability behavior. Finally, future work will be discussed and conclusions will be given.

2 Characteristics of a Sports Management Resource Planning System

In this section we will describe the primary characteristics of an SMRP system and distill a list of requirements that constrain the design of our architecture

A sports association is a central organization, whos main operational purpose is organizing sports matches. A sports match typically involves teams in which the players are organized. A team is part of a sports club. Hence, the club is the entity in which teams are organized. The players of a team are also members of the corresponding club. Organizing a sports competition involves several tasks: registering teams, administrating members, setting up match fixtures, assigning

officials, informing several stakeholders about match results/schedules/changes, administrating match results, processing financial tasks, recording- and acting upon disciplinary events, organizing courses, managing venues, supporting clubs with their internal organization and many more. Each of these tasks involves several user roles. Some of these roles are implemented within the central sports association, the degree of which is dependent on the level of professionalism in the sports association. Larger sports associations tend to have a higher level of professionalism. However, some roles are logically out-sourced to other organizations like the sports clubs, but also to individuals like e.g. match officials. The logical out-sourcing is based on the following adagium: Keep information close to its source. The difference in relevant skills, between an individual working professionally for a sports association and an individual working voluntarily for a small sports club, can be very large. This has implications for how each of these different users can optimally interact with the SMRP application.

In addition, sports associations, partly due to their differences in level of professionalism and partly due to historic reasons, can not be treated completely uniformly from the point of view of a generic information system. Internal regulations can imply completely different business rules when considering e.g. disciplinary checks, financial handling, member administration. Also, different third party applications maybe used within a sports association that need to be integrated with the SMRP system. Typical third party applications are web applications that are used to implement an association's website, an increasingly important part in the operation of any modern association. Information available on web sites, related to schedules, match results, etc. are directly obtained from the SMRP system. Integrating with third party web applications exposes the system to a possibly hostile environment, the Internet. Considering expected system usage, the largest sport association in the Netherlands hosts a web site that has 100.000+ visitors each day. Since an SMRP system lives in the 'leisure' domain, the usage pattern is not constrained to working days/hours, but in fact the system is heavily accessed by non-professional users outside office hours.

The amount of knowledge accumulated and built into the system during the lifetime of an SMRP system like Sportlink is very large. Therefore, it is very expensive to rebuild a new SMRP system from scratch. Taken together with the fact that most sports associations do not have enough financial resources for building a new SMRP system, has been the reason for many sports associations to opt for a SaaS solution. Also, rebuilding the entire SaaS solution using a new technology is not opportunistic given the small financial margins in the Sportlink commercial subscription fees. Hence, the system needs to be relatively insensitive for becoming outdated. This is not an issue for the core of the system, i.e. the data model and the algorithms are quite insensitive for becoming outdated. But, it is especially in the visible parts, i.e. the user interfaces, and higher level functionality of the application that the system does suffer from becoming outdated. User interfaces are subject to a certain amount of technological volatility. Especially technological developments within the browser and the mobile device domains are progressing quite fast. In addition, the degree of

digital matureness is different between sports associations, but also progressing with time. Currently not all potential digital processes have been completely implemented. However, it is necessary that new automated business processes can be implemented easily.

Having summarized the characteristics of the SMRP system, we can now derive the following list of architectural requirements for our application:

1. Heterogeneous *usability* requirements: the system needs to be operated by both professional as well as non-professional users with a wide range of relevant skills. Hence, there is an expected wide range of functionality requirements for user interaction. Professional users are typically using the system during a full working day, while non-professional users are more likely to use the system occasionally.
2. Heterogeneous *integration* requirement: arbitrary functionality can be integrated with arbitrary different other information systems.
3. *Multi-tenant* requirement: different business rules, depending on the sports association, will have to be implemented using a single application. Hence, the system should be easily configurable without creating clones of the system that need to be adapted (possibly resulting in maintainability problems).
4. *Modification* requirement: there is slow but sure tendency within sports associations to further mature the degree of automation. Newly automated businesses processes should be implemented relatively fast and without compromising the stability of existing functionality, i.e. a high level of agility is required.
5. *Durability* requirement: low cost extension of the application lifetime is needed.
6. *Stability* requirement: 24x7 stable operational system and 1M+ web service requests per day in a 'hostile' Internet environment have to be supported.
7. *Latency* requirement: since the system is also used 'interactively' by end-users following a request-reply pattern, response times for those services should be within reasonable bounds.
8. *Scalability* requirement. The amount of system users is theoretically unlimited. Potentially every member of a sports association can be a user of the system, requiring a scalable solution..

In the next section we will choose the architectural styles needed to fulfil our requirements.

3 Choosing architectural styles

Separation of concerns (SoC), originally coined by Edsger W. Dijkstra[7], is a well-known concept in Computer Science. It means breaking up a program in non-overlapping functionalities (separation) and focusing on each (concern). This is a good and proven software engineering practice and can be applied also to a web application architecture. In that context, it means breaking up a

web application into *layers* that are functionality non-overlapping. One, usually, recognizes three tiers: *presentation-*, *logic-* and *data* tiers.

Many web applications are build according to this three tier architecture. In the case of a J2EE web application, a domain model is implemented using (Enterprise) Java Beans. Possibly some Object-Relational-Mapping framework is used to persist relevant Java Beans. The user interface is implemented using some popular web application framework based on e.g. Java Server Pages (JSP) technology[2]. The entire end user community uses a single web based application in which functionality can be hidden/shown based on user roles. In the event that some functionality of the system needs to be integrated with third party applications, a WSDL/SOAP interface is typically generated from Java Classes using popular web service frameworks like Apache Axis[3]. We call the application stack that we have just described, *a monolithic application architecture*.

However, in our case such an architecture has serious limitations. A monolithic application approach will not be appropriate for the following requirements: usability (1), integration (2) and durability (5). Besides horizontal, layered SoC, we also need SoC in other parts of the architecture. Let us consider requirement (1). It will be impossible, or at least very difficult, to implement a single web application that can be used for the large variety of users of the SMRP system. Also, a web application is not very well suited for implementing highly interactive, fast responding, user interfaces, needed to support the association employee. In addition, classic web applications do not particularly scale very well. This is due to their inherent client-server duality. Part of a web-application lives in a habitat called a browser, part of a web application lives in a web server. The memory or the state of a web-application resides on the web-server. Hence, for each web application session, resources need to be allocated on the web server. Rich desktop clients, backed by a stateless service layer, have much better scaling characteristics. On the other hand, due to zero installation overhead, web applications are better suited to support a large set of anonymous users that also require limited application functionality.

Requirement (2) dictates that any functionality of the system should be available to arbitrary third party systems. In a monolithic application approach, third party application integration is always treated separately on a need-to-use basis. In our design we consider requirements (1), (2), (5) and (8) together by using a layer consisting of, *stateless*, loosely coupled services that serves all the different end-user applications as well as any third party application. The usability and integration requirements can be covered by the introduction of a *functionally complete service layer*: the entire functionality of the system is represented by services. Both third party applications as well as front-end applications are consumers of these services. By also shifting the responsibility of *state* to those applications we can employ a stateless and scalable service layer. The durability of our application is increased by putting all business functionality into services that can be invoked both separately and in some orchestration. Basically, every

[2] http://java.sun.com/products/jsp/
[3] http://ws.apache.org/axis/

businesses process is represented by a service. The SMRP application as a whole becomes durable by the disposability(!) of the end user applications, enabled by seperating end-user application and business process concerns.

There are other requirements to consider for our SMRP application. A purely Object Oriented architecture does not very well capture the multi-tenant (3) and modifiability (4) requirements. A pure OO approach is too rigid and tightly coupled to support easy modification. Modifying the application, by introducing new functionalities, requires building and deploying an entirely new version of the application. Also, implementing slightly different business processes for different associations requires re-coding in the case of OO for even the most simple changes. By constraining the use of pure OO only in the rigid, core parts of the system, e.g. those parts that rely heavily on complex data manipulation used in planning algorithms, we can effectively isolate and decouple atomic system parts. By encapsulating those atomic parts and integrating them, as required, using flexible service definitions, we are introducing a Service Oriented loosely coupled architectural style that does fit the multi-tenant (3) and modifiability (4) requirements.

Lastly, the stability requirement (6) is poorly represented in a synchronous layered architecture, as a consequence this may lead to unexpected latencies (7) and poor response times. Peak loads or hostile attacks could lead to degenerate system stability and even crashes. In addition, resources that are accessed by the system could misbehave temporarily. In a purely synchronous, layered, architecture, this can cause a performance loss also for services that are not dependent on the misbehaving resource. A *Staggered Event Driven Architecture* (SEDA)[9], yet another kind of SoC, can help solve this problem. To further enhance the stability of the system we need an independent software layer for observing the state of the system and execute corrective actions. Such a de-coupled software layer in itself needs to be extremely reliable and stable, *autonomous software agents*[13] can provide these characteristics.

The architecture that we envision is in fact a modern variant of *component-based client/server computing*[11]: *service-based client/server computing*.

4 The Sportlink Software Architecture

In this section we will describe the architecture and frameworks that are used to implement the Sportlink system.

The Sportlink system can be considered as an application ecosystem with a service based core (depicted by the 'Services' layer of Fig. 1). It is not a single monolithic application, but in fact composed of multiple (end user) applications each serving different stakeholders within and among the different sports associations (the 'Applications' layer in Fig. 1). The service layer, implemented using the *Navajo framework*[4], delivers all the necessary services for the different end-user applications, as well as for third party application integration. Each of the

[4] http://www.navajo.nl/doku.php

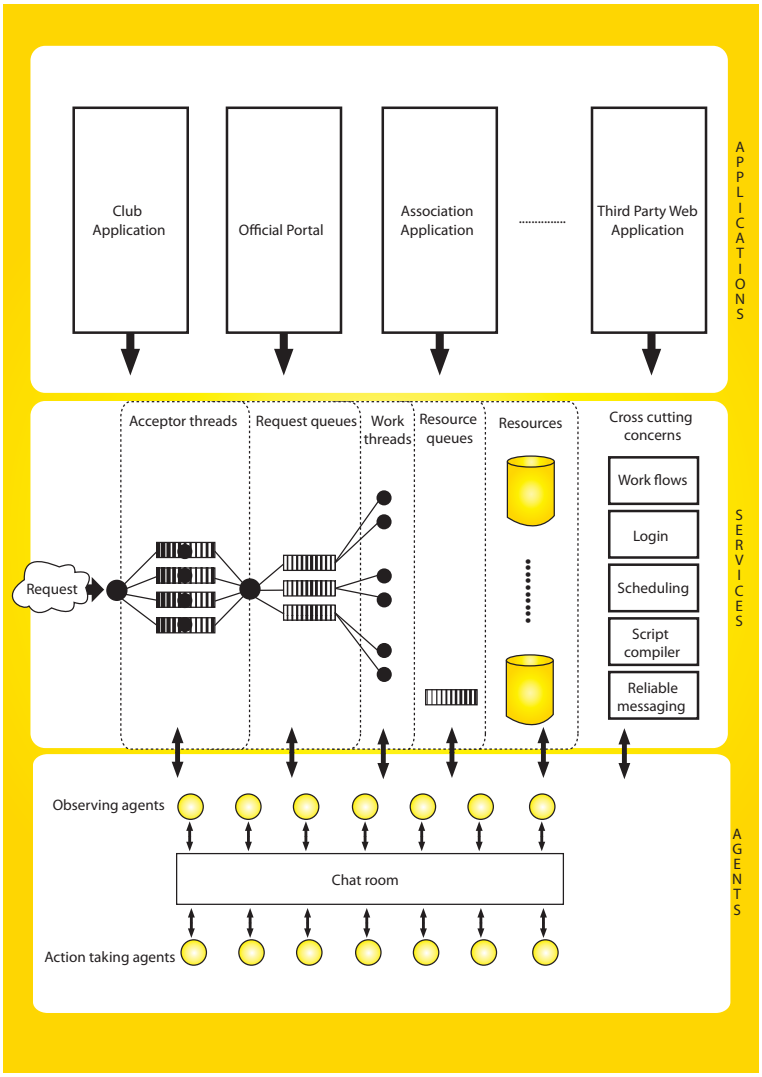

Fig. 1. Architecture of the Sportlink system showing the three primary layers: Applications, Services and Agents

1,800 different web services, is implemented using a flexible scripting language. A script instance can be versioned for the purpose of configuring association specific functionality (requirement (3)). The scripting engine supports hot deployment enabling the introduction of new web services without any application downtime.

We are very much in favor of contract-first service development [10]. However, according to current standards we would have to create at least three software

artifacts for each set of services: a WSDL file, an XSD definition and the actual code that implements the service. The Navajo scripting language (Navascript) combines all three of these, still being able to derive a WSDL and an XSD definition if required. A script consumes and produces a request/response in an internal XML format[5]. An XML request can originate from several (protocol) sources like: XML over HTTP, a SOAP web service call but also via a chat XMMP protocol[6] from a Jabber server. The script specifies how the request is processed and how the XML response is produced. An analogy can be made with an XSLT transformation. However, Navascript is much more constraint and focused, because it can only consume and produce the native Navajo XML format.

Although our scripting language enables flexible functionality and hot deploy, a more *aspect-oriented* methodology is sometimes preferred [6] . We support event-based *service workflows* which are based on service invocation triggers that define the join points to introduce cross-cutting behavior. Cross cutting functionality is in turn implemented as a web service. Examples of cross cutting functionality that are currently in use: (1) web service caching and cache flushing, (2) audit logging, (3) third party application integration and (4) human interaction involved business workflows. A workflow consists of states to define cross cutting functionality and transitions to define state (or context) dependent join points. Each workflow has a unique init state for bootstrapping a workflow instance. In addition a unique finished state must exist. A service workflow is defined using XML. Workflows enable even more flexible, association specific functionality without compromising stable, core, functionality.

Special attention needs to be given to web service communication latencies. Using a simple synchronous, blocking I/O servlet approach, we have observed certain anomalies in service response times which are hypothetically attributed to Java Garbage Collect freezes during peak traffic and misbehaving resources. We have solved this issue by using non-blocking I/O, effectively decoupling thread creation from incoming service requests. To decouple request traffic from doing the actual work for a service, we have introduced several worker thread queues. Different queues are used both for implementing a quality of service for first-class service consumers and for parking service requests that are dealing with temporarily badly performing resources (e.g. database systems). Using this approach, we can handle external, misbehaving, service requesters and resources without compromising system response times (requirement 7).

A service layer simply offering a lot of functionality would not be very useful without any clients consuming its services. The diversity of stakeholders and their diverse interaction requirements has led to the implementation of a variety of end user applications, even end user applications that were specifically built for a single sports association (requirement 4). Consider for example individual members that require personalized match information and functionality to update personal data. Other users, like officials, use mobile devices during a match

[5] http://www.navajo.nl/navajodocument/spec.html
[6] http://xmmp.org/

for which temporary application failure due to network problems is intolerable, i.e. a semi-offline application is required. It follows that we need a plethora of end-user applications using a variety of technologies to support different usability constraints (the 'applications' layer in Fig. 1). Efficiently building user interfaces on top of the service layer is an important requirement for the Sportlink architecture. To support this requirement we have built an Abstract User Interface Definition framework, *Tipi*, in a similar spirit of the work found in [3] and [2]. Many Sportlink end-user applications use Tipi-script, a declarative programming language for developing end-user applications, which is *front-end-technology-agnostic*. Hence, applications can be deployed using several front-end solutions. Services are the underlying model for all of these applications. Implementing business logic using Tipi-script is constraint by only supporting user-interface and service-invocation/listening/manipulation semantics. Currently, Tipi implementations exists for Java/Swing and HTML/Javascript (employing the Echo Framework[7]). For single, non-orchestrated, sequential web service interactions, a Tipi client exists that automatically creates a user interface with support for several data types to UI widget mappings, including multi media types. Automatically generating a user interface is enabled by the Navajo XML language that offers support for UI annotations like (multi lingual) descriptions, types (including drop down lists and binary formats) and additional meta data for input validation and mime types.

A *Pluggable Agent based Monitoring* (PAM) backbone (depicted by the 'Agents' layer of Fig. 1) is used for establishing a self healing eco-system[5] and thereby facilitating requirement (6). The management backbone consists of observing- and action-taking agents that use the XMMP protocol for communication, a chat room is where the agents meet. Specialized observing agents monitor the state of system components (like databases, access logs and application servers) in order to publish their findings in a chat room. Action-taking agents roam the chat room for messages and participant states, discovering system failures, abuse, mis-configurations, performance bottlenecks, etc. Whenever malicious patterns are discovered, either autonomous corrective actions are undertaken by those agents or system operators are noticed using an appropriate messaging system (e-mail, SMS, Twitter). Agents can (and for stability must) be deployed anywhere, as long as they can subscribe to the chat room and can connect to the system components (Observing Agents) they represent.

5 Feasibility of the solution

The Sportlink SMRP system has been implemented using the described architecture. Since 2004, Sportlink is actively used by 10 different sports associations representing approximately 2 million members. Every week, 50,000 matches are played. Volunteers at 7,000 sports clubs are actively accessing the system using a specialized Tipi/Java Swing based club application (see Fig. 2). Approximately

[7] http://echo.nextapp.com/site/

10,000 match officials use a tailor made web portal that provides a self-service application and more than 1,000,000 players/supporters actively visit association, and club-websites to check current standings. The Sportlink system infrastructure consists of three application servers running Tomcat backed by three Oracle database servers. We have observed web service loads of 200 services / second, which the system can easily process. Recently, the service load has been increasing due to a number of newly automated processes. The scalability of the system enables adding an additonal application- and (read-only) database-server.

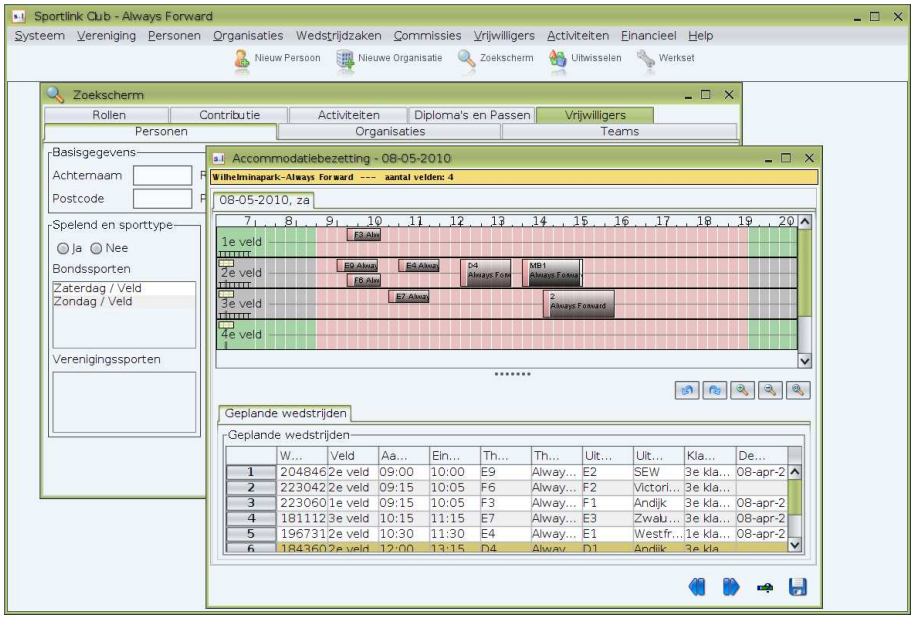

Fig. 2. A screen shot of the club application, showing a venue planning window

We have used our Tipi framework to build a number of different applications: a club application (screenshot shown in Fig. 2), an official portal, a venue management system, a desktop digital match form application and a Sportlink Management Console. Each application has been tailormade for a specific user group. Although, rich desktop user interfaces have been created with Tipi, the development team was mostly recruited from developers with minimal web development skills. Seperation of Concerns could be applied within the development team by identifying front-end- and service developers. Due to the fact that the structure of Navajo services support automatic generation of user interfaces, service developers can work almost independently from front-end developers.

To give an indication of the typical load of the system, Table 5 depicts the distribution of (client side) response times. The data has been collected during

a period of three months using a service access logging system that employs delayed logging of client side response times (since these times are not known when processing the original service request). The diversity in the response times is rather large due to the amount (1,800) of different web services in use and their broad range of functionality. According to our statistics, 1,400 different services take less than 500ms to complete, representing 86% of the total service load. Almost 50% of the web services are processed in less than 100ms. This last category is mainly composed of web services for populating tables, text- and drop down boxes with data. The other categories typically represent services for populating large tables and data-/CPU intensive calculations.

Response time bandwidth	*Unique services*	*Total requests*
0 - 100ms	1,327	49,398,217
100 - 500ms	1,301	39,847,849
500 - 1000ms	725	10,516,431
1000 - 5000ms	580	3,800,658
5000ms+	339	95,246

Table 1. Response time distribution over unique services and total service calls during a three months logging period. Note that the same service could fall in several interval bins due to response time variation.

Although there are large differences in business rules and application integration requirements between the 10 different sports associations, the Sportlink system is still a single source application, clearly showing the validity of the multi-tenant requirement. All of these differences are implemented using authorization rules, validation rules, service workflows, and to a small (less than 0.1%) extent association specific versions of web services.

Sportlink offers a specialized club application (see Fig. 2), however, one association has chosen to use a variety of club applications that are available on the open market. For member administration consistency, these application need to be integrated (member mutations) with Sportlink. The integration was entirely implemented using service workflows within a time frame of only a couple of weeks from project start to production. This can be considered quite an achievement since our solution replaced a Message Broker system that took over a year to build. Furthermore, due to the approach of using loosely coupled workflows, the modification did not impact other functionality whatsoever.

Since the PAM backbone is still under development, we can not report on any recent experience figures related to the improved stability of the system.

6 Future work

In this section we will describe the direction in which the architecture of Sportlink is evolving.

In our opinion, optimal user experience is a very important asset for successful software applications. The advent of the classic web applications in the late '90s and early '00s was actually a step backwards in terms of application usability. There was a major rush towards *thin (web) clients*[12], in order to centralize control of applications and to relieve the burden of systems administrators. Most of the time this was at the expense of the user experience. Web applications are limited by the browser, and back then that was very limited. Since then web applications have gradually been catching up with technologies like AJAX, JavaScript, and recently HTML 5. Another interesting recent phenomenon is that there has been a revival of client side applications driven by mobile platforms and the gaming industry (*iPhone AppStore*, *Google Marketplace*, *Steam*). Using online App Stores eliminates the installation, update and security woes that fueled the initial exodus towards web applications. Whatever the future may bring, we believe that we need a technology agnostic layer for clients, which describes the structure and interaction of a client, without the technical details of actually rendering the UI on the client. This is actually the same Separation of Concerns we described before, also applied to the client. This way we also have a multi-tier client. In fact, our vision of client interaction far transcends the technical decision of thick or thin clients. For us, the client, and even the user, is part of the service oriented architecture. The traditional model of applications assumes that the user is in control of the entire application. This makes complete sense for offline applications, but for a dynamic ecosystem that is simply not true. Events occur, and the UI needs to deal with it. Who triggered it or why is irrelevant. This is an essential and often overlooked aspect of a truly service oriented system. For example, the system may need a user to make a decision. Perhaps a specific user, perhaps any qualified user. In a simple case, the system may choose to send an notification to a user, post a message to a chatroom, or send a text message, requesting the user to take action. In this case the interaction is still not full circle: notification emails are old news. We can however push a disposable client to the user that is specific for that occurrence. It takes the form of mailing an HTML form to a user, or maybe an url to a web application. Only now the users are truly part of the ecosystem: Users can request services of computers, and computers can ask services of the users. Our goal is a more task driven approach for interacting with the system.

Other future work includes implementing a semantic model, combining both a Navajo- and a domain ontology for supporting a tool, a Software Information System (in the spirit of [8], but by using modern semantic technology), that can be used to estimate the impact of e.g. system changes.

7 Conclusion

It is not dynamic-run time-binding-of-components-, or inter-enterprise integrations requirements that led to the Service Oriented architecture of the Sportlink System. Our requirements are a little bit less ambitious: a strict separation of core (configurable) functionality and front-end applications to support agility

and system durability. We have shown that these requirements can be a motivator for employing a service oriented development approach as opposed to a traditional monolithic methodology.

We hypothesize that applications following the SaaS (Software as a Service) model are typical candidates for applying the same methodology as we have described for our Sport Management Resource Planning System. We have developed a service oriented development framework, Navajo, that can be used as a concrete implementation of our suggested service oriented approach. In addition to a feature rich service run-time, we have developed a service oriented declarative user interface frameworks, Tipi, that can be used to define event driven abstract user interfaces backed by a Navajo service layer. In combination, both frameworks have proven to satisfy all characteristics of a large scale enterprise information system.

References

1. Robinson, G.: Kotonya: A Pluggable Framework for Tracking. In: ICSOC 2009. LNCS vol. 5900, pp. 637–638. Springer, Heidelberg (2009)
2. Kassoff, D. Kato, W. Mohsin: Creating GUIs for web services, IEEE Internet Computing, pp. 66–73, IEEE (2003)
3. F. Paterno, C. Santoro, L. D. Spano: MARIA: A Universal, Declarative, Multiple Abstraction-Level Language for Service-Oriented Application in Ubiquitous Environments
4. M. Papazoglou: Service -Oriented Computing: Concepts, Characteristics and Directions. In: WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering, IEEE (2003)
5. M. Papazoglou: Service-Oriented Computing: State of the Art and Research Challenges. IEEE Computer November 2007, pp. 64–71, IEEE (2007)
6. J. Niemoller, R. Levenshteyn, E. Freiter, K. Vandikas, R. Quinet, I. Fikouras: Aspect Orientation for Composite Services in the Telecommunication Domain. Service Oriented Computing: Proceedings of the 7th International Joint Conference, ICSOC-Service Wave 2009. LCNC, vol. 5900, pp. 19-33, Springer Heidelberg (2009)
7. Dijkstra, Edsger W.: On the role of scientific thought, in Dijkstra, Edsger W., Selected writings on Computing: A Personal Perspective, pp.60–66., New York, NY, USA: Springer-Verlag New York, Inc.
8. Devanbu, P. T. and Brachman, R. J. and Selfridge, P. G. and Ballard, B. W.: LaSSIE—a knowledge-based software information system. In: ICSE '90: Proceedings of the 12th international conference on Software engineering, pp. 249-261, IEEE Computer Society Press (1990)
9. Matt Welsh: An Architecture for Highly Concurrent, Well-Conditioned Internet Services. Ph.D. Thesis, University of California, Berkeley, August 2002
10. Hillenbrand, M., Gotze, J., Muller, P.: Contract-first service development within the Venice service grid. In: iiWAS '08: Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services, pp. 48-54, ACM (2009)
11. Lewandowski, Scott M.: Frameworks for component-based client/server computing, ACM Comput. Surv. Vol. 30(1), pp. 3–27, ACM (1998)

12. Fulton, J. and Kramer, E.: Can you ever be too thin?, netWorker Vol. 1(2), pp 19–23, ACM (1997)
13. Hyacinth S., Software Agents: An Overview, Knowledge Engineering Review, Vol. 11(3), pp 1–40, Cambridge University Press (1996)