Solent — a Platform for Distributed Open Hypermedia Applications

Siegfried Reich[1], Jon Griffiths[1], David E. Millard[1], and Hugh C. Davis[1]

Multimedia Research Group
University of Southampton
SO17 1BJ, UK
E-Mail: {sr, jpg96r, dem97r, hcd}@ecs.soton.ac.uk

Abstract. Today's open hypermedia systems (OHS) provide middleware services for a range of hypertext applications. However, configuration and adaptation to specific applications' requirements is a tedious task. Research has been conducted into further splitting hypermedia middleware systems up into sets of interacting components that can be combined, extended and configured dynamically. These component-based open hypermedia systems (CB-OHS) allow for better adaptability, configurability and also interoperability amongst hypermedia middleware systems themselves.
Described is the *Solent* component-based open hypermedia system. In particular, we focus on architecture, dynamic service discovery and invocation as well as the storage interface, which allows for storage and retrieval of arbitrary hierarchical structures encoded in XML.

Authoring and managing links are without doubt essential issues in hypertext systems. Thus, research has been conducted into separating link data from content, i.e. the documents, which results in link services being available to *all* applications on the user's desktop. These third generation hypertext systems are called *open* and are referred to as open hypermedia systems (OHS) [18].

There are several consequences of designing a system with separate links [10]: documents can still be managed and edited by those applications that created them and more links can be applied to documents that are read-only, such as documents on a CD-ROM or documents owned by another user. In addition to this, multiple users can use their own link bases at the same time, thus there are different views on the same information space. One of the most important features lies in the possibility of link maintenance: when updating links only the link bases have to be updated, the documents themselves remain unchanged. This clearly has great advantages in contrast to systems with embedded links — the WWW serving as a very prominent example — because these systems demand all relevant documents to be changed.

While the development from the early monolithic and closed hypertext systems to today's open and distributed hypermedia systems with support for collaboration has resulted in increased openness and flexibility [13, 20], integrating or adapting various different tools as client applications has remained a tedious

task. Many developers found themselves implementing essentially similar components, simply for the benefit of having their own platform on which to experiment with hypertexts.

At the Second Workshop on open hypermedia systems (OHS) held in conjunction with the '96 ACM Hypertext Conference [19] the open hypermedia systems working group (OHSWG) was formed, with its main focus being interoperability between OHSs [18]. The group felt that the community had reached a level of maturity and stability such that it was possible to abstract the common features of the various systems, and to propose to move towards one of the major goals of any open system: interoperability.

This goal resulted in the hypermedia middleware, i.e. the link servers, being split up into several components with well defined interfaces [2, 20]. Some of these interfaces, e.g. for navigational hypertext, have been standardised by the open hypermedia systems Working Group (OHSWG) and therefore allow interoperability amongst components [7].

This paper describes the design and implementation of the Southampton component-based open hypermedia system (CB-OHS) which we refer to as the "Solent" system, named after the waters around Southampton. The system has been developed to address the many issues posed to hypermedia middleware and serves as a platform for the development of advanced multmedia applications such as content-based retrieval [4], content-based navigation [12], or navigation in audio [3].

1 Requirements

The objective of efficiently supporting a diverse range of advanced open hypermedia applications by a set of middleware components leads to a number of requirements that have to be met. Besides the more technical requirements, there are goals that we as a research group would like to address, such as building a framework that could serve as a research vehicle with which we could further experiment.

Dynamic Setup of Components (Requirement 1, R1). This precondition arises from our experience in providing middleware services for various open hypermedia applications. Configuring these applications is often a quite complex task, additionally it may require updates to components at runtime with minimal impact on other components, dynamic lookup of services, and more [8]. Besides, within the context of the research undertaken towards interoperability amongst open hypermedia systems, there is the need to support multiple hypermedia link models at the same time and allow dynamic exchange and adaptation of these. Often components run on various platforms and therefore the communicaion mechanism between components should be platform independent.

Support for Navigational Hypertext (R2). Various forms of navigation still build the core features of many hypertext systems. Thus, it has been a

requirement to support the standardised navigational interface as promoted by the OHSWG [7].

Support for Storage of Arbitrary Structured Data (R3). The data being dealt with in hypermedia applications is often highly structured. Hence it is a requirement to support such structures, e.g. encoded in eXtensible Markup Language (XML) we believe that support of this standard is necessary. XML has an interesting property in that it allows a mode whereby no document-type-definition is required to form an XML document (hence approaches such as [5] cannot be directly applied). This implies that the storage system has to support storage and retrieval of structures that it does not "know" about. Besides encoding of data in documents, XML is also often used for communicating messages between components. This further strengthens the need for support of XML (see also Section 2.2).

Support for Computations (R4). In particular for supporting content-based retrieval [4] and navigation in multimedia documents [12], it is necessary to support the abstract notion of a computational service as a "black-box" which takes defined parameters as input and produces a fixed output. A computation stores the information about a service in the same way that a node represents a wrapper to a document, it is a meta-information object.

Open, Extensible Framework (R5). It was a requirement to build a framework which can be used as a research vehicle in a number of research projects. By *Framework* we refer to a software environment that is designed to simplify the development and management of hypermedia applications [2] by supporting re-use of existing components [11]. In particular, we want to research issues such as streaming content data and its synchronisation with link data, the use of computations for content-based retrieval and navigation, and adjust the architecture to different configurations in order to e.g. address scalability [1].

2 Description of the Architecture

The architecture of the *Solent* system builds on the experience gained in developing the open hypermedia systems Microcosm [6], Microcosm TNG [9], MEMOIR [8] and others. Figure 1 depicts the conceptual architecture.

The Figure shows the separation of components into front-end components (i.e. the applications), middleware services, and the storage backend. This is in agreement with both other CB-OHSs such as Microcosm TNG [9] or Construct [20] and also the requirements mentioned above (R1, R5).

Solent is designed to support the standardised interfaces being developed within the OHSWG [7], distributing that support over many communicating components. Key to the system is the notion of an "engine" which is a software process managing a certain functionality within the system. The most essential

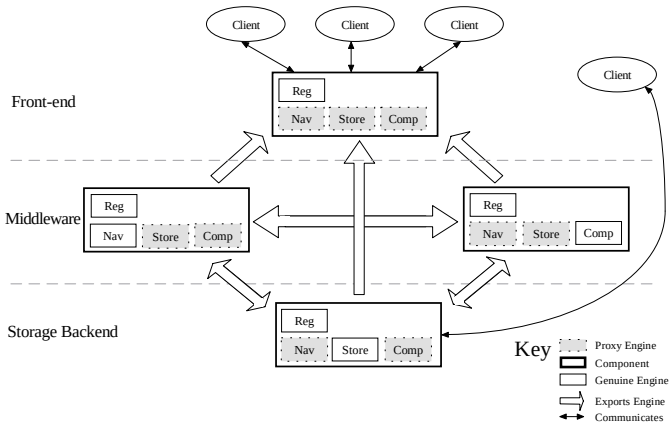

Fig. 1. The Component Architecture of the *Solent* System

engine, present in every component, is the *registration engine* ("Reg" in Figure 1). The purpose of this engine can be compared to that of an information broker, e.g. in Corba. This engine is tightly coupled to the hosting component. Its job is to receive registration requests from other components in the system. When one component registers with any other it sends its connection details (the protocol and version it speaks as well as the host and other location information that it resides on). The registration engine then creates a *proxy engine* (see grey boxes in Figure 1) in its own component and sends its own registration information back. Registration and de-registration can be truly dynamic (R1, R5).

A proxy engine appears to the outside world as would any other engine but in actual fact proxies all requests to the real engine in a separate component. This has several advantages:

1. When a component registers it sends details not only of its own genuine engines but also to all of the proxy engines.
2. Because the proxy engine retains the connection information of the genuine engine, when it registers it sends this information rather then that of the proxy. This means that messages are only ever re-directed once (client to proxy to engine).
3. Any component can be treated as if it contained an appropriate engine as long as it knows the location of such an engine in another component.

The communication objects are currently configured to work using messages represented in XML over plain TCP/IP sockets as "on-the-wire" protocol. Other communication mechanisms such as Corba's IIOP or Java's RMI have been in-

vestigated as well; however, due to reasons of platform independence and standardisation within the OHSWG, XML over plain Sockets has remained the communicatin mechanism of choice. Each incoming message is offered to each engine in the component, it is up to the engine to decide if it wishes to deal with a particular request. Each engine does this based on the protocol and version of the message being sent (if it can differentiate between them) and also the name of the engine that the message was intended for. As a result, concurrent support of multiple versions of a particular interface is possible (R1, R5).

This architecture results in a group of communicating components that can be dynamically configured and distributed over multiple platforms and at the same time act together cohesively as a whole. Even though currently only two core hypertext interfaces are supported, namely the navigational and the computational interface (R2, R4), the architecture allows for an easy integration of interfaces to other hypertext domains such as spatial hypertext [14] or workflow applications [17].

In the following subsections we will describe two components in more detail, the computation and storage components.

2.1 Dynamic Service Discovery and Execution

Hypermedia systems often allow their proprietary clients access to advanced functionality, in particular for supporting retrieval [4] and navigation within multimedia documents [12]. To enable this functionality to be accessed by generic clients it is necessary to support the abstract notion of a *computational service* (R4). A computation is a black box of functionality, known to a client only by its name as well as its input and output parameters, that can be invoked and its results understood, even though its workings are completely opaque. In this way a generalised client gains access to complex functionality that would otherwise be unavailable (see also Section 3).

The OHSWG has been working on a standard interface for accessing these computations. The *Solent* system contains a computation engine that understands what computations are and the interface that deals with them.

Often computations may actually take a long time to complete resulting in the obvious problem of giving feedback to the user. The definition of the computation interface therefore includes a mechanism which lets the calling component know the time that a service takes and allows the called component to send progress messages back to give explicit feedback. This can be compared to the notion of "Quality-of-service" (QoS).

Sometimes a single computation cannot effectively represent a facility. For instance, for content-based navigation in audio [3], a computation might analyse a section of music and produce a *contour*; it would then check that contour against the ones in a database to produce a list of nodes representing musical scores similar to the selection. The issue here is that although the input and output parameters of this complex computation are well defined the computation itself needs to be split up into several sub-computations for scalability and performance reasons. Clearly, the analysis of a huge audio file should take place

at the file's location, however, matching the resulting contour against a database could be performed elsewhere. We have therefore come up with the definition of a composite computation.

Composite computations represent a way for components to "impart knowledge" to the rest of the system about how computations can be combined (in serial or in parallel). This leaves plenty of scope for the development of more sophisticated systems which learn about particular combinations of computations and also provides a framework for the exploration of mobile code and agents within a CB-OHS environment.

2.2 A Storage Engine for XML

The *Solent* CB-OHS contains a storage component which corresponds to the *storage engine* of Figure 1 (R3). Its role is to manage the storage of arbitrary structured data encoded in XML. The storage engine comprises a relational database which other components of the *Solent* system can access via a Java application, called the *storage manager*.

We have found that being able to store arbitrary structured data has afforded some benefits.

- It encourages rapid prototyping of protocols which describe meta-data about components of the system, such as the navigational or the computational interfaces. This is because different versions of the same protocol can be stored in the same database at the same time, without having to create a new database or alter the schema of existing database tables.
- The ability to store arbitrary structured data enables the installation of new components within the system without having to consider the protocol that is being used to manipulate the meta-data representation of its objects, the only factor being that the meta-data must be able to be represented in XML.

Due to the fact that all (structured) objects are stored as flat relations in the database, retrieving objects from the database involves building XML elements which are composed of smaller entities: sub-elements and values. The XML elements used to represent the arbitrary structured data in the storage engine take the form of n-ary trees; i.e., each XML sub-element can be considered to be a branch and XML values represent leaves of a branch.

The implementation of the storage engine comprised of several steps, these were particularly necessary due to performance issues. We will now briefly describe these steps.

Firstly, data was retrieved using a *breadth-first* approach. This method begins at the element root and works its way down the tree. A typical retrieval operation using this technique took 121 seconds.[1] The problem with this approach is that

[1] The data was calculated with the following configuration: Pentium P75, 32 MB Ram, Windows 95, JDK 1.1, MS-Access 95. Our test data consisted of around 80 high-level objects which resulted in approximately 2500 XML objects stored in the database overall. Times given are for a typical query with a single user.

when storing similarly structured objects in the database, the difference between objects meant for retrieval and ones that are not, may only be perceivable when examining the values of the leaves of the retrieved XML structure. Therefore there may be a lot of retrieval activity within the database, which is retrieving structure that is not associated with the database objects meant for retrieval, but this is not discovered till late into the retrieval process. This is an inherent flaw when retrieving XML structure using a top-down approach, where the discerning information is located at the bottom of the structure.

In a next step, data was retrieved using a *depth-first* approach. This involved retrieving the value at the botton of the branch for each unexplored sub-element of an element. A typical retrieval operation using this method took 28 seconds. A major problem concerned with this approach is when there are lots of objects in the database which match the object meant for retrieval. By retrieving objects from the bottom-up, the retrieval algorithm may begin constructing objects which comprise the same structure as one of those objects meant for retrieval, but may actually contain the values and/or sub-elements which belong to different objects. Therefore such objects should not be retrieved after all.

Thirdly, we investigated a *combination* of depth-first and breadth-first retrieval methods. It follows the same steps as the depth-first retrieval technique until the first branch is encountered within the template object. Then instead of searching the database for the value at the end of that branch, it performs the breadth-first technique for retrieving database entries down that branch. The performance measured 24 seconds on average for a typical retrieval operation.

In a final step, we employed *caching* of objects in main memory to further increase the efficiency of retrieving objects from the database. On system start-up, each object stored in the database was built into its corresponding XML structure. Then when an object was requested, a breadth-first retrieval algorithm was employed for object retrieval on the database cache. Although breadth-first retrieval had proven to be the most inefficient retrieval method earlier, caching easily compensated for this inefficiency. A typical retrieval operation took 5 seconds.

We also investigated query optimisation at the application level [16]. This involved modifying the client application which sent database retrieval requests. The order of the elements which the storage engine would use as a basis for searching the database was changed in order to optimise the retrieval operation.

In general, our approach towards flexibility by supporting storage and retrieval of arbitrary structures encoded in XML meant that we needed to investigate various tuning techniques to achieve the performance needed for our applications.

3 Sample Applications: Generic Media Player and Car Stereo

A number of hypermedia applications have been built using the *Solent* system. We will briefly describe two examples, a software Car Stereo and a generic Media

Player. The Car Stereo Viewer replicates a car CD system which can play files encoded in the MP3 standard and it was built specially for use within the *Solent* system. The second application, a generic media player, is an adapted client of the windows media player which has been developed for content based navigation in audio [3]; it demonstrates how services can be dynamically discovered using the compu

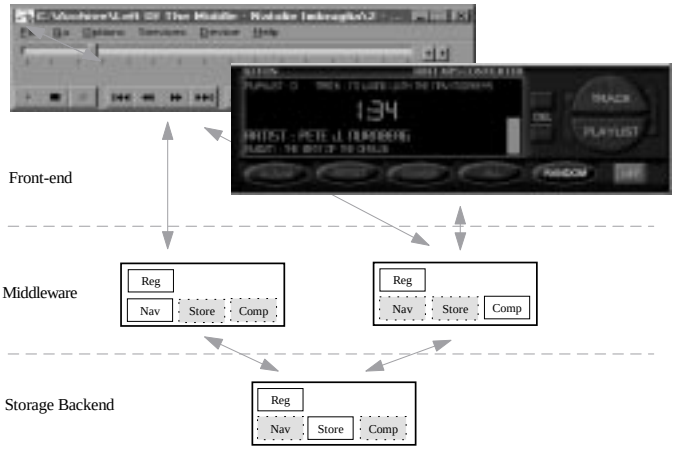

Fig. 2. Media Player and Car Stereo as Sample Applications of *Solent*

As can be seen in the Figure, the adapted generic Media Player (see top left) communicates using two interfaces, the navigational and computational interfaces, whereas the purpose-built Car Stereo client only uses the computational interface. The Car Stereo has a built-in set of requests it understands, those can be instantiated by users through pressing a button. E.g., users might want to ask for "similar songs" to the one that they are currently listening to.

The Media Player on the other hand, is a generic application that, besides others, supports the computational interface. Hence, this application is able to negotiate with a computational engine the set of services it is offered. The Media Player understands the basic mechanisms on how to call a computation and how to parse its results, i.e. it knows how to deal with input and output parameters. In doing so it can dynamically discover the set of computations available, offer them to the user, perhaps ask the user to provide it with some additional parameters and finally execute the computation. By supporting the navigational interface the Media Player not only allows users to retrieve tracks by executing a computation, but also to navigate from e.g. a track to some description in form of an HTML file on the Web, etc. This clearly demonstrates the advantage of component-based open hypermedia systems in that specific parts of functionality can be supported and combined if feasible.

4 Summary and Conclusions

In this paper we have considered the evolution of open hypermedia systems to component-based open hypermedia systems. In particular, we have described the design and development of the *Solent* CB-OHS which have been driven by the requirements being set by today's advanced hypermedia applications.

The experiences we gained during the design, implementation and prototyping are manifold.

- The modular design clearly helped in restructuring the system to the different application areas and allowed us to better select the components to support various functionalities [15].
- Furthermore, the flexibility helped to address the problems encountered in serving different versions of protocols and interfaces that we needed to support during the development of standardised interfaces within the OHSWG's interoperability effort [7].
- The modularity of the system allowed us to run different interfaces simultaneously assisting interoperability, in particular at the front-end [20].
- The trade offs between flexibility and performance have been shown by the example of the storage layer. The feature of being able to support arbitrary structures proved useful during the development phase with structures changing constantly and also, often being undefined (i.e., no document-type-definition existed). On the other hand, for demonstrating the system as described above it was necessary to tune the performance of the retrieval operations.

In summarising we believe that the current system provides us with a platform which allows us to exploit research issues of open hypermedia systems. In particular, we are interested in addressing scalability aspects of our architecture not only with respect to data [1] but also with respect to number of users, number of concurrent computations, etc. Additionally, we are currently investigating the feasibility of a standardised interface to access documents within open hypermedia systems. This includes issues such as streaming document content, synchronising content data with link data and more.

Acknowledgements

The authors would like to thank the members of the multimedia research group at Southampton, in particular Steve Blackburn, Dave DeRoure, Ian Heath, and Wendy Hall. We would also like to acknowledge the members of the open hypermedia systems community, in particular Ken Anderson, Pete Nürnberg and Uffe Wiil.

References

1. Kenneth M. Anderson. Data scalability in open hypermedia systems. In *Hypertext '99, Darmstadt, Germany*, pages 27–36, February 1999.

2. Philip A. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2):86–98, February 1996.
3. Steven G. Blackburn and David C. DeRoure. A tool for content based navigation of music. In *Multimedia '98, Bristol, UK*, pages 361–368, September 1998.
4. Stavros Christodoulakis and Peter Triantafillou. Research and development issues for large-scale multimedia information systems. *ACM Computing Surveys*, 27(4):576–579, December 1995.
5. Vassilis Christophides, Serge Abiteboul, Sophie Cluet, and Michel Scholl. From structured documents to novel query facilities. In *SIGMOD '94, Minneapolis, Minnesota*, pages 313–324. ACM, 1994.
6. Hugh C. Davis, Wendy Hall, Ian Heath, Gary J. Hill, and Robert J. Wilkins. Towards an integrated information environment with open hypermedia systems. In *Hypertext '92, Milan, Italy*, pages 181–190, 1992.
7. Hugh C. Davis, David E. Millard, Siegfried Reich, Niels Olof Bouvin, Kaj Grønbæk, Peter J. Nürnberg, Lennert Sloth, Uffe Kock Wiil, and Kenneth M. Anderson. Interoperability between hypermedia systems: The standardisation work of the OHSWG (technical briefing). In *Hypertext '99, Darmstadt, Germany*, pages 201–202, February 1999.
8. Dave C. DeRoure, Wendy Hall, Siegfried Reich, Aggelos Pikrakis, Gary J. Hill, and Mark Stairmand. An open architecture for supporting collaboration on the web. In *IEEE WET ICE 98, Stanford University, California*, pages 90–95, 1998.
9. Stuart Goose, Jonathan Dale, Gary J. Hill, Dave C. DeRoure, and Wendy Hall. An open framework for integrating widely distributed hypermedia resources. In *Multimedia '96, Hiroshima*, pages 364–371, June 1996.
10. Wendy Hall. Ending the tyranny of the button. *IEEE Multimedia*, 1(1):60–69, 1994.
11. Ralph E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, October 1997.
12. Paul H. Lewis, Hugh C. Davis, Steve R. Griffiths, Wendy Hall, and Robert J. Wilkins. Media-based navigation with generic links. In *Hypertext '96, Washington, D.C.*, pages 215–223, March 1996.
13. Peter J. Nürnberg, John J. Leggett, and Uffe K. Wiil. An agenda for open hypermedia research. In *Hypertext '98, Pittsburgh, PA*, pages 198–206, 1998.
14. Olav Reinert, Dirk Bucka-Lassen, Claus A. Pedersen, and Peter J. Nürnberg. CAOS: A collaborative and open spatial structure service component with incremental spatial parsing. In *Hypertext '99, Darmstadt, Germany*, pages 49–50, February 1999.
15. Douglas E. Shackelford, John B. Smith, and F. Donelson Smith. The architecture and implementation of a distributed hypermedia storage system. In *Hypertext '93, Seattle, WA*, pages 1–13, 1993.
16. Dennis E. Shasha, *Database Tuning: A Principled Approach*. Prentice Hall, 1992.
17. Weigang Wang and Jörg M. Haake. Implementation issues on ohs-based workflow services. In Uffe Kock Wiil, editor, *5th Workshop on Open Hypermedia Systems, Hypertext '99, Darmstadt, Germany.*, pages 52–56, 1999.
18. Uffe Kock Wiil. Open hypermedia: Systems, interoperability and standards. *Journal of Digital Information (JoDI). Special Issue on Open Hypermedia*, 1(2), 1997.
19. Uffe Kock Wiil and Serge Demeyer, editors. *Proceedings of the 2nd Workshop on Open Hypermedia Systems, Hypertext '96, Washington, D.C.*, 1996.
20. Uffe Kock Wiil and Peter J. Nürnberg. Evolving hypermedia middleware services: Lessons and observations. In *ACM Symposium on Applied Computing (SAC '99), San Antonio, TX*, pages 427–436, February 1999.