The Impact of Research on the Development of Middleware Technology

Wolfgang Emmerich
Dept. of Computer Science, UCL, Gower Street, London, WC1E 6BT, UK

and

Mikio Aoyama
Nanzan University, 27 Seirei, Seto 489-0863, Japan

and

Joe Sventek
Dept. of Computing Sciences, University of Glasgow, Glasgow GG12 8QQ, UK

The middleware market represents a sizable segment of the overall Information and Communication Technology market. In 2005, the annual middleware license revenue was reported by Gartner to be in the region of 8.5 billion US Dollars. In this article we address the question whether research had any involvement in the creation of the technology that is being sold in this market? We attempt a scholarly discourse. We present the research method that we have applied to answer this question. We then present a brief introduction into the key middleware concepts that provide the foundation for this market. It would not be feasible to investigate any possible impact that research might have had. Instead we select a few very successful technologies that are representative for the middleware market as a whole and show the existence of impact of research results in the creation of these technologies. We investigate the origins of web services middleware, distributed transaction processing middleware, message oriented middleware, distributed object middleware and remote procedure call systems. For each of these technologies we are able to show ample influence of research and conclude that without the research conducted by PhD students and researchers in university computer science labs at Brown, CMU, Cambridge, Newcastle, MIT, Vrije, and University of Washington as well as research in industrial labs at APM, AT&T Bell Labs, DEC Systems Research, HP Labs, IBM Research and Xerox PARC we would not have middleware technology in its current form. We summarise the article by distilling lessons that can be learnt from this evidenced impact for future technology transfer undertakings.

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Software Architectures; D.4.4 [Operating Systems]: Communications Management; H.2.4 [Database Management]: Systems; D3.3 [Programming Languages]: Language Constructs and Features; K.1 [The Computer Industry]: ; K.2 [History of Computing]:

This report was written as part of the ACM SIGSOFT sponsored Impact project, which was funded jointly by The Institution of Electrical Engineers and the National Science Foundation.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20YY ACM 0000-0000/20YY/0000-0002 $5.00

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20YY ACM 0000-0000/20YY/0000-0002 $5.00
1. INTRODUCTION

Various commercial trends have led to an increasing demand for integrated systems. Firstly, the number of mergers and acquisitions of companies is rising. The different divisions of a newly merged or acquired company have to deliver unified services to their customers and this usually demands an integration of their IT systems. The time available for delivery of such an integration is often so short that building a new system is not an option and therefore existing system components have to be integrated into a system that appears as an integrating computing facility. Secondly, time to market requirements are becoming more stringent. Increasingly the only approach to meeting delivery schedules is the procurement of components off-the-shelf and their subsequent integration into a distributed system. Components to be integrated may have incompatible requirements for their hardware and operating system platforms; they might have to be deployed on different hosts, forcing the resulting system to be distributed. Finally, the Internet provides new opportunities to offer products and services to a vast number of potential customers. In this setting, it is difficult to estimate the scalability requirements. An e-commerce site that was designed to cope with a given number of transactions per day may suddenly find itself exposed to demand that is by orders of magnitude larger. The required scalability cannot usually be achieved by centralized or client-server architectures but demands a distributed system architecture.

Distributed systems can integrate legacy components, thus preserving investment, they can decrease the time to market, they can be scalable and tolerant against failures. The caveat, however, is that the construction of a distributed systems is considerably more difficult than building a centralized system. This is because there are multiple points of failure in a distributed system, system components need to communicate with each other through a network, which complicates communication and opens the door for security attacks. Middleware has been devised in order to conceal these difficulties from application engineers as much as possible; Middleware is commonly defined as a software layer between applications and operating systems that provides application programmers with higher level of abstractions, such as remote procedure invocation, reliable message exchange or transactions. These abstractions considerably simplify distributed system construction and as a result middleware products are rapidly being adopted in industry [Charles 1999] and middleware is generally perceived as a success technology.

Though there is anecdotal evidence for middleware successes, it is very hard to quantify how successful middleware technology really is. An indicator might be how widely used middleware products are. Again those data are hard to obtain as the fact that someone has downloaded or installed an open source middleware or bought a license for a middleware product does not necessarily mean they are actively using it. However it is not necessarily required to have precise data here to get an impression and license revenue may serve as a useful approximation. A recent Gartner study by [Correia and Biscotti 2006] has determined that the world-wide market for middleware and application integration products has grown in 2005 to 8.5 billion US Dollars in annual license revenue. Figure 1 shows an overview of the market share that various middleware vendors had in 2005.

When interpreting these numbers, it is worthwhile to bear in mind that the overall economic activity that critically depends on middleware is considerably larger for a number of reasons. Firstly, there are a significant number of deployments of open source middleware, such as JBoss [Fleury and Reverbel 2003]. According to the Gartner study JBoss...
had 10 million installations in 2005 but JBoss is available free of charge and it therefore is not reflected in Figure 1. Secondly, in addition to license revenues, middleware vendors have often significant revenues in professional services, which are not included here. Thirdly, the total significance of the application server market is larger yet as IBM, BEA, Oracle, Microsoft and Tibco rely on third-party integration service providers to deliver the bulk of their professional services and finally, the full economic significance of middleware can only be understood when we also take into account the wealth that distributed applications whose construction has been enabled with middleware create in domains such as e-commerce, finance, entertainment and education.

The principle contribution of this article is a scholarly account of the impact that research has had on the development of middleware technology. We undertake an existence proof rather than attempt a comprehensive account of any influence that research results might have had. Our key findings, which will be refined further in this paper, are shown in Figure 2. The figure uses a notation that we will define more formally in Figure 3.

We document that distributed transactions, which are crucially important for any Java application server product, are standardized in the Java 2 Transaction API, which in turn builds on the CORBA Object Transaction Service. The CORBA OTS was defined by a team that involved Graeme Dixon, who completed a PhD thesis on object transactions in the Arjuna project at the University of Newcastle. The OTS also defines the concept of Nested Transactions, which were defined in the PhD thesis of Elliot Moss at MIT.

A second constituent specification for any J2EE application server is the Java Messaging Service (JMS) that enables asynchronous and reliable exchange of messages between distributed Java objects. JMS standardized interfaces to message queuing products and we are able to trace these products back to research undertaken by Skeen at Teknekron and to work done on the Broadcast Message Service that Steven Reiss devised for tool integration in the Field software development environment.
The Java RMI specification is again crucially important for any application server to enable synchronous communication between Java objects that reside in different Java Virtual Machines. RMI was defined at Sun by a team led by Jim Waldo. In an early paper describing RMI, Waldo identified the key influences that Modula-3 Network Objects had on RMI. Network objects were defined by Andrew Birrell and Greg Nelson Digital System Research. Birrell and Nelson state in their paper that they have merely consolidated and simplified earlier research results and refer to a number of systems, including Emerald, whose type system was defined in Hutchinson’s PhD thesis and by Orca, whose object model was defined in Bal’s PhD thesis.

A similar impact trace that features Andrew Birrell’s earlier work exists for Remote Procedure Call systems, that were standardized by the IETF. The ONC Request for Comment refers to the work by Birrell and Nelson at Xerox as the origin of the concept. Bruce Nelson had previously completed his PhD on RPC systems at CMU and it was rendered usable when Xerox built the Cedar programming environment. For the purpose of defining RPC interfaces, the Cedar RPC system used a language called Mesa, which had abstractions for module interconnection purposes.

Finally, the latest form of middleware, such as Enterprise Service Buses, uses web service technology, which implements the World-Wide-Web Consortium’s SOAP and WSDL specifications. Don Box, the first author of the first W3C note on SOAP, gave a historic ac-
count on the origins of SOAP and he acknowledges that the team have used concepts from RPC systems and CORBA. A similar statement exists for WSDL in a paper co-authored by Francisco Curbera, who is a co-author of the WSDL specification. Both SOAP and WSDL are XML languages and XML traces back to the ISO SGML standard, which was defined by a team involving Goldfarb. Goldfarb had earlier defined IBM’s Markup Language GML and in a retrospective paper on GML identified Scribe as giving the key ideas. Scribe was defined as part of Brian Reid’s PhD thesis at CMU. The first CORBA specification was written by Joe Sventek and Jim Waldo while they were at HP Labs (and before Waldo went on to work at Sun). Sventek had earlier worked on the ANSA middleware, which was very influential on the ISO/Open Distributed Processing Reference Model. ANSA extends an RPC model with object-oriented concepts and acknowledges Birrell and Nelson’s work on RPCs.

This paper is further structured as follows: We will discuss the research method that use in this report in Section 2, which includes identifying the different classes of evidence that we admit for proving impact. We introduce the notion of impact traces and the graphical representation used in Figure 2, which relate these pieces of evidence to show the history of how a particular concept got introduced into a product or standard. In Section 3, we will present the main middleware concepts that are necessary to follow the discussion. In Sections 4-8, we will present the impact traces that show the relationship between research results and successful middleware products. In Section 9, we discuss lessons about industrial adoption of research that can be learnt for future technology transfer and Section 10 concludes the paper.

2. RESEARCH METHOD

We note that the nature of this research is considerably different from those normally published in ACM Journals. This project reflects on the influence that research results of the past have had on middleware technology. The project is therefore more akin to the discipline of history of science than software engineering research and in many ways it increases the level of reflection in that research itself is now the subject of our investigation.

Since the research method we apply may be novel to readers, we think it appropriate to describe the method first. It was influenced by sibling working groups in the Impact project with whom we have exchanged our experience and informed by the project’s History of Science expert, but there are several aspects that are special to the relationship between middleware technology and academic and industrial research, our subject of interest.

In this section, we first address the different forms of evidence that we are prepared to admit to document that a method, technique, specification or idea has influenced something else. We then introduce the notion of an impact trace that captures the trail from a successful standard, product or method back to its origins in research. We finally discuss the different methods that we have used for finding evidence of influence or impact.

2.1 Evidence of Impact

Gartner’s recent report confirms that the middleware market is significant. At the same time it is quite fragmented and the different technologies we consider here are in different fragments of the middleware market. It is therefore insufficient to just state how large the overall market is but we have to determine the market relevance of the particular technology under consideration. To perform that assessment, we use evidence extracted from market studies conducted by professional market research firms, such as Forrester, Gartner, Ovum
and IDC. The aim of these firms is to systematically advise investors and decision makers in the IT industry on current market trends, including the quantitative value of particular market segments. Although they have to be interpreted carefully, they are useful to give an indication on how intensively middleware is being used in practice. Once evidence for an existence of a middleware market has been established, the next set of questions that needs answering is from where companies obtained the ideas, languages, methods, techniques and principles that they use in middleware products and services.

When inventing a new technique, idea or method companies often take steps to protect their intellectual property. They do so using national or international patents. The patent offices typically force companies to disclose prior art in the area and explain the relationship between the new invention and the prior art. Fortunately, these patent documents are publicly accessible and form important pieces of evidence.

Vendors of middleware products and services do not encourage scholarly discipline for acknowledging the origin of ideas in the same way as academics do in scientific publications. On the contrary it is often advantageous for career development if engineers do not disclose the origin of ideas in writing. In order to trace the source of ideas we will also have to admit oral testimonials of engineers, who made key decisions on building products. We therefore record and archive each interview with these engineers to provide this evidence.

An important vehicle for the exchange of ideas between research institutions and vendors of middleware products and services is the movement of people who know about these ideas. Together with testimonials, we are therefore also prepared to admit evidence of impact if someone, who completed a piece of research in a university or industrial research lab, moves on to a middleware product or service vendor and provides a recorded testimony of what ideas they have taken along and used in their new job.

Fortunately, we do not just need to rely on such oral accounts. Companies form de-facto standardization bodies and contribute intellectual property to the standardization effort. In fact, the middleware industry is heavily driven by both de-jure and de-facto standards, such as the Common Object Request Broker Architecture of the Object Management Group, ISO’s Open Distributed Processing standard and the Java Community Process that adopted various middleware standards. There are certification authorities, such as the OpenGroup that formally certify that particular products comply to a standard and we take such certification as evidence that the product complies with and therefore uses the main ideas and principles that have been incorporated in the standard. Even if no formal certification is available but if companies claim that they comply to the standard we are prepared to use that as evidence that they have used the ideas embodied in the standard when developing their products.

Standards may have reference implementations. These reference implementations are sometimes provided by research institutions. We regard the exchange of such reference implementations as important evidence for impact. More generally, the use and inspection of prototypes built in research institutions by industrial middleware vendors is significant. Such exchange is generally achieved by downloads from ftp or web servers and we will be looking for the download logs to provide evidence for such use of prototypes and reference implementations.

The documents that define the standard themselves lay out the principles, ideas and techniques used in the standard and sometimes they explicitly reference the source of those ideas. We consider these as very significant pieces of evidence for the sake of this study.
However, absence of any citations in a standard documents does not necessarily mean that there was no impact.

Standardization bodies provide another source of evidence that we use. In particular, the standardization process of the OMG and the Java Community Process are open processes and they are minuted to record the decisions that have been made together with their rationale. We are therefore also prepared to admit the minutes of these meetings as evidence for this study.

Moreover, the communication in international standardization bodies is typically widely distributed and therefore heavily driven by electronic communication. In many cases, this communication is also archived for future reference. We are prepared to admit email exchange and notes posted to bulletin boards of standardization bodies if they can be attributed to individuals and explicitly reference the origin of an idea, principle, method or technique, possibly in an academic paper.

Citations of academic papers are among the strongest form of evidence that we can find for the purpose of this study. We will look for those citations in textbooks that are used to teach students, professional books, standards, case study reports and white papers and in scientific papers themselves. The academic papers that we are prepared to admit are refereed publications, either in national or international conferences, symposia or workshops or in scientific journals published by either professional societies, such as the ACM, IEEE or IEE or by professional publishers, such as Wiley or Kluwer.

2.2 Impact Traces

The Impact project aims to show how research influenced products and industrial best practices. [Redwine and Riddle 1985] showed that it takes a significant period of time for software engineering research results to be incorporated into products and industrial best practices. In most cases, impact of research that was published in an academic journal or presented at an international conference will not be directly picked up by some product development teams but its influence will be more indirect.

The aim of this study is therefore to relate individual pieces of evidence into an impact trace, which uses the different forms of evidence discussed above to show the research origins of successful middleware products and best practices for their application. In most cases, these impact traces will not necessarily be linear lists, but rather a directed acyclic graph. It is important to align impact traces in time to form a timeline, which gives the direction of edges in the graph. In this report we show these traces graphically in graphs with time progressing from bottom to top. We differentiate between the different forms of impact by using different lines as shown in Figure 3.

The impact traces also follow certain patterns. In particular, near the root there are indications of evidence on how successful a particular product or method is. Evidence related to standards or applied research papers will be found in the middle of these traces, while leaves will have to be scientific publications in journals and conferences or PhD theses.

We note that it is difficult to relate middleware to a particular computer science research discipline. Middleware does not occur in the ACM classification of computer science and is therefore not (yet) a discipline in its own right, even though conferences that focus on middleware have been created in the last ten years. We expect impact traces for middleware to cross boundaries to computer science disciplines and in particular between programming languages, databases, distributed systems and software engineering research. Given the
sponsorship of this paper by ACM SIGSOFT and its publication in a software engineering journal, we are particularly interested in impact of software engineering research. Indications for whether an article or paper describes “software engineering research” may be drawn from co-sponsorship of the conference where the paper was presented. If the conference is sponsored or co-sponsored by the IEEE Technical Council of Software Engineering or the ACM Special Interest Group on Software Engineering or if the editorial board of a journal is largely composed of members of either of these groups then this can be taken as strong evidence that the paper describes software engineering research. On the other hand papers might be published by general journals, such as CACM or IEEE Computer.

2.3 Finding Impact Traces

Now that we have discussed the different kinds of evidence that we are prepared to admit as evidence for impact and introduced the notion of impact traces, we can focus on how we obtain impact traces in a systematic manner.

A main source of impact will be references in research papers published in journals and conference proceedings. While the authors and contributors to this report have a reasonable grasp on the middleware literature, purely relying on this knowledge would deliver only partial results. In order to construct paths in impact traces related to citations in academic papers, we are therefore going to rely on citation databases, such as Google Scholar, Citeseer or the ACM Portal to Computing Literature. The advantage of this approach is that it directly reveals impact traces. The disadvantage is that the knowledge base of Google Scholar, Citeseer or the ACM Portal to Computing Literature, while continuously improving, is still not complete.

A more complete knowledge base digital libraries and portals, such as those provided by IET Inspec, SpringerLink, ACM and the IEEE. Information retrieval in these databases are now accessible via the Web and candidate papers can then be downloaded in electronically. The ability to search in Portable Document Formats, the most commonly used representation in digital libraries greatly improves our search for impact traces.

Information retrieval techniques can also be used to obtain impact traces in the mailing list archives and meeting minutes of standards bodies, such as the OMG. In particular, the OMG minutes are accessible by search engines, such as Google and Google can be
constrained to only search, for example the OMG web site.

The above techniques will not support finding impact traces that involve evidence that is not generally accessible, either because it is not available on a public web site or because it may not even be accessible in an electronic form. We will therefore also revert to discussions with the original engineers, who were involved in building the successful projects. Any such discussion will be recorded in order to make the evidence mentioned verbally tractable.

3. MIDDLEWARE CONCEPTS

This section introduces all important middleware concepts that the reader who is not necessarily familiar with distributed systems will require to understand the remainder of this article.

The first mention of the term ‘middleware’ with the meaning that it has today was made by Alex d’Agapeyeff at the 1968 NATO Software Engineering Conference in Garmisch Partenkirchen. On Page 23 of [Randell and Naur 1969], d’Agapeyeff is reported to have stated that middleware is layered between applications and “service routines and control programs”, which we would today refer to as operating systems. At the time of the NATO conference and for a long time thereafter, d’Agapeyeff was the managing director of Computer Analysts & Programmers in London. Anthony Hardcastle, one of d’Agapeyeff’s employees gave a talk at a 1972 meeting on “Developments in Computer Auditing” of the London and District Society of Chartered Accountants. The subject of Hardcastle’s talk was “file design and organization, programming and testing” and during the talk Hardcastle explained the term “middleware”. A brief summary of the talk was published in the journal Accountant [Jenkins 1972]. The article states: “A comparatively new term ‘middleware’ was introduced because, as some systems had become ‘uniquely complex,’ [sic] standard operating systems required enhancement or modification; the programs that effected this were called ‘middleware’ because they came between the operating system and the application programs”. The Accountant article [Jenkins 1972] is referenced by the Oxford English Dictionary as the etymological origin of the term “middleware”.

The term middleware was made popular in the computer science literature in a CACM article by [Bernstein 1996]. In-line with d’Agapeyeff’s and Hardcastle’s definition, the term ‘middleware’ today is regarded as a layer between network operating systems and applications that aims to resolve heterogeneity and distribution [Bernstein 1996; Emmerich 2000]. A distributed system has one or more component on more than one host. For the system to appear as an integrated computing facility these components need to communicate with each other [Colouris et al. 1994]. In theory, components could communicate by directly using the primitives that network operating systems provide. In practice, this is too complex as application programmers would have to synchronize the communication between distributed component that execute in parallel, transform application level data structures into byte streams or datagrams that can be transmitted by network transport protocols and resolve the heterogeneity of data representation on different hardware architectures. The main aim of middleware is to simplify these tasks and provide appropriate abstractions that application programmers use when building interacting distributed system.

---

1We have written to the editors of the Oxford English Dictionary to suggest that Alex d’Agapeyeff’s talk at the NATO Software Engineering Conference be cited as the etymological origin of the term middleware rather than Chartered Accountant meeting summary by [Jenkins 1972].
components. We now define the most successful abstractions that are widely supported by middleware products.

The first key middleware abstraction that was defined in the distributed systems literature are remote procedure calls (RPCs). The principle idea of a remote procedure call is to enable application programmers to invoke a procedure that is deployed, and executes, on a remote host in a similar way as they would invoke a local procedure. In order to achieve this, remote procedure call systems have to accomplish a number of tasks. Firstly, the RPC would need to map complex application level data types that are passed as parameters and results into a transport representation that can be transmitted using the byte stream or datagram transport protocol primitives that network operating systems typically provide. This mapping process is referred to as marshalling in the literature. Secondly, the RPC system might have to transform data representations as the hardware, operating system and programming language on the client and server host cannot be assumed to be the same. For example, alphanumeric data may use different encoding schemes, such as ASCII or EBCDIC, mainframes and personal computers use different number representation schemes, different programming languages represent character strings differently, and so on. Resolution of data heterogeneity is typically achieved by RPC systems through mapping to and from a standardized serialization format, sometimes also referred to as a standardized data representation. Thirdly, the client that invokes the RPC and the server that performs the RPC are generally executing in parallel. In order to mimic the behaviour of local procedure calls that block the caller until the called procedure terminates, RPC synchronization blocks the RPC client until the remote procedure call either completed or failed. Finally, due to the existence of multiple points of control, failures in distributed systems are more likely to occur. RPC systems therefore need to define a richer failure semantics.

Figure 5 shows the common aspects of the architecture of most RPC systems. The central architectural element are stubs, which perform synchronization, marshalling, mapping into a standardized data representation and the network communication. These stubs are sometimes also referred to as proxies as they represent proxies of the server objects for clients and proxies of the client objects for servers. One of the key insights of early RPC systems was that it is actually hard to build stubs manually and it became common to generate them automatically. RPC systems typically define some form of interface definition language (IDL) that is used to determine the signatures of remote procedure calls. RPC
systems come with an IDL compiler that is then used to generate stubs automatically. In some approaches the interface definition language may be separate of a programming language, in which case a *programming language binding* needs to be defined that determines how IDL language primitives are represented in the programming language.

Since RPCs were first introduced the, key principles, most notably the synchronous invocation of an operation on a remote host have been reused in different circumstances. *Distributed object systems* were defined as an object-oriented extensions of RPC systems and their interface definition languages include primitives to express inheritance, exception handling, late binding and polymorphism. The invocation of a remote invocation in a distributed object system is often referred to as an *object request*. Modern programming languages, such as Java, now natively support *remote method invocations* that are essentially RPCs of methods that are hosted in another Java Virtual Machine. Finally, web services evolved from RPCs by using XML for defining a service interface definition language and also as a standardized data representation.

Once remote procedure call systems were gaining popularity, application designers had to build systems that were reliable, fault-tolerant and concurrently usable. A key abstraction that aided in this respect is the notion of a *distributed transaction*. Distributed Transactions are sequences that typically contain more than one remote procedure call. Transactions are atomic, consistent, isolated and durable. Transactions had been defined and used earlier [Gray 1978] in database management systems, but the distributed transactions required middleware support as server component might work on different databases. These abstractions are commonly provided by transaction processing middleware that is sometimes also referred to as transaction processing monitors (TPMs) [Bernstein 1990]. These TPMs implement a two-phase commit protocol that allows transactional resource managers, such as databases and message queues to first decide whether they are able to commit a transaction and once all participating resource managers agree to commit the TPM then asks them to perform the commit.

Remote procedure calls and transactions are useful abstractions when interaction between distributed system components is fundamentally synchronous. There are also situations when an asynchronous interaction style is more appropriate. For example, when distributed systems operate in different time zones and are not guaranteed to be available it might be preferential to buffer communication. Likewise, architectures that have high
throughput requirements often demand buffering of requests in order to ensure constant utilization of processors. Such asynchronous interaction is today mostly implemented using message passing abstractions that are provided by message oriented middleware (MOM). Message-oriented middleware provides message queues through which messages can be sent asynchronously and reliably between distributed system components. Message queues can persistently store messages so that they survive failures of the messaging middleware or the receiver. Message queues typically support point-to-point communication. Message queues also support distributed transactions so that messages can be enqueued and de-queued under transaction control. Message-oriented middleware products often also implement publish/subscribe based communication that permits one message to be sent to multiple recipients.

Message queues, remote procedure calls and transaction middleware have been integrated in application server products. Application servers provide a distributed component model that supports the remote invocation of operations. The life cycle and activation of distributed components is often controlled by a container. When the first remote call to a component is made the container activates the component, i.e retrieve the component state from persistent storage and transfers it into a state so that it can serve the call. In modern application servers these requests can be made synchronously using a remote procedure call or they can be triggered asynchronously when a message arrives on a message queue. Application servers also support the notion of transactions and the container might start and terminate distributed transactions on behalf of every component operation that is executed so that application programmers do not have to implement transaction protocols themselves. Application servers are to date frequently employed in the implementation of service-oriented architectures. We now discuss the origins of this technology.

4. MIDDLEWARE FOR SERVICE-ORIENTED COMPUTING

Since its debut in 2000, Web service technologies have been evolving at an astonishing speed. A number of specifications related to Web services and Service Oriented Architectures (SOA) have already been standardized and an even larger number are currently under development. Since Web services are platform technology, the impact of Web services to software development practice is tremendous. A good introduction into web services technology is provided by [Weerawarana et al. 2005].

The core technologies supported by any web services offering are the SOAP\footnote{SOAP used to be an acronym for Simple Object Access Protocol. However, in the latest W3C specification SOAP is used as a noun in its own right.} [Box et al. 2000; Gudgin et al. 2003], the Web Services Description Language (WSDL) [Christensen et al. 2001; Chinnici et al. 2004] and the Business Process Execution Language (BPEL) [Andrews et al. 2003]. It is these technologies that the majority of service oriented architectures in enterprises build upon. We now trace the origin of these middleware technologies but first again sketch their relevance.

SOAP and WSDL received instant enthusiasm when they were first introduced in 2000. However, due to their very nature as platform technologies, it required a few years to develop supplemental technologies such as the sizable Web Service Security specifications [Nadalin et al. 2004], as well as to gain enough technical maturity and expertise to develop distributed applications using Web services. To date Web services provide a universal and common infrastructure for building distributed systems. Such infrastructure is
indispensable to federate applications across organizational boundaries. Web services are now implemented by all application server products, including Microsoft BizTalk server, BEA's Web Logic Server, IBM's Websphere, Oracle Application Server, JBoss and various other open source offerings. The resulting impact of Web services is tremendous. According to a Forrester Research survey, some 50% of large enterprises are using Service Oriented Architectures (SOA) now or will use it by the end of 2006 [Heffner et al. 2006]. It should be noted that Web service technologies are not a mere computing platform. They have started to change software business modeling, from selling software to selling services over the Web, and the way of developing a business. Thus, the impact of Web services and SOA are beyond computing [Hagel and Brown 2002; Bieberstein et al. 2005]. We now discuss the origins of the main constituent specifications that form the basis of web services technology.

4.1 Simple Object Access Protocol

SOAP is a standard messaging format based on the eXtensible Markup Language (XML). SOAP provides a mechanism to represent an object invocation and its response in XML messages that are then transmitted using an underlying network transport protocol. SOAP can be used with bindings for both synchronous and asynchronous protocols, such as HTTP or SMTP. Thus, SOAP is independent of transport protocols. SOAP supports two messaging styles, RPC (Remote Procedure Call) and document-literal. We show the impact traces that led to the definition of SOAP in Figure 6.

SOAP is often implemented in a web server or a web container, which includes Apache Axis and Microsoft IIS (Internet Information Services). Application servers that provide direct support for SOAP, which include BEA Web Logic Server, Fujitsu Interstage Application Server, IBM WebSphere Application Server, IONA Artix, and Microsoft BizTalk Server. Implementations of tool kits for processing SOAP messages include the Apache Axis toolkit available in Java and C++, the IBM Web Services Toolkit, and the Microsoft SOAP Toolkit. There are also several open source SOAP processors implemented in Perl, PHP and Ruby.

The currently valid and widely used definition of SOAP in Version 1.2 was adopted as a World-Wide-Web Consortium Recommendation in June 2003. The SOAP development started in 1998 and SOAP went through a number of revisions. The evolution of the SOAP specification is documented by the first author of the initial Note published by the World Wide Web consortium on SOAP in [Box 2001]. In this report, Box states essentially that SOAP emerged directly from earlier RPC protocols by pointing out that “we looked at the existing serialization formats (ASN.1 BER, NDR, XDR, CDR, JRMP) and RPC protocols (GIOP/IIOP, DCE/DCOM, RMI, ONC) and tried to hit the sweet spot that would satisfy the 80% case elegantly but could be bent to adapt to the remaining 20% case.” Thus Box states that SOAP has been influenced by, and been built upon, remote procedure call and distributed object technologies. We will discuss the origins of these distributed object and RPC technologies in Sections 7 and 8.

4.2 eXtensible Markup Language

SOAP uses the eXtensible Markup Language (XML) [Bray et al. 1998] as the encoding for requests and reply data that is transmitted between the client and the server involved in a web service invocation. A language for defining XML type systems was not well elaborated when the SOAP effort started. In particular, XML did not yet have a schema
definition language and XML schema, which the final version of SOAP uses for definition of data types. XML schema was only finally defined in 2004 [Birron et al. 2004]. The absence of a schema language for XML stalled the SOAP standardization somewhat and an interim approach known as XML/RPC was defined by [Winer 1999] and obtained rapid adoption. When SOAP was finally released it supported both RPC-style invocations as proposed by [Winer 1999] and document-literal invocations that involve passing typed XML messages that encode request and reply data.

The XML family of specifications that include XML [Bray et al. 1998], XML Schema [Birron et al. 2004] and XPath [Clark and DeRose 1999] were adopted by the World-Wide-Web Consortium. XML forms the basis for SOAP, WSDL and BPEL. XML evolved from the ISO Standardized General Markup Language [ISO 8879 1986]. SGML.
emerged from, and was defined by the same team that designed IBM’s General Markup Language [Goldfarb 1981], whose development started at the IBM Cambridge Scientific Centre that was embedded within MIT. [Goldfarb 1981] acknowledges that [Reid 1981b] had developed the notion of procedural markup in the Scribe document processing system. Scribe is fully described in [Reid 1981a]. The central idea of GML that survived in SGML and XML is the strict separation of concern between content and presentation. Separation of concern is not particularly novel for software engineers and it is therefore not surprising that Goldfarb uses the production of Cliff Jones’ software engineering textbook [Jones 1980] in GML as a case study in [Goldfarb 1981].

4.3 Web Services Description Language

WSDL provides the language abstractions to define interfaces of web services. It supports the identification of port-types, which correspond to modules or types in other IDLs, the identification of operations and operation signatures. There is an intricate relationship between SOAP and WSDL in that parameter types used in WSDL interfaces determine the XML documents that can be used within SOAP messages. We now discuss the origins of WSDL. The impact traces that led to WSDL are visualized in Figure 7.

Many application servers and software development environments support WSDL. Apache Axis provides bidirectional proxy generators between both Java and C++ and WSDL. Most application servers come with Web services development tools. IBM Rational Application Developer for WebSphere generates a WSDL document from existing applications and Java proxies from a WSDL document. Microsoft Visual Studio generates
proxy for .NET Framework from WSDL document. Other tools include BEA Workshop for WebLogic Platform, Fujitsu Interstage Apworks, and IONA Artix.

Ariba, Microsoft and IBM jointly issued a W3C Note in March 2001 defining a Web Services Description Language [Christensen et al. 2001]. The Note states “this draft represents the current thinking with regard to descriptions of services within Ariba, IBM and Microsoft. It consolidates concepts found in NASSL, SCL, and SDL (earlier proposals in this space)”.

The Network Accessible Service Specification Language (NASSL) was defined by [Curbera et al. 2000] at the Component Systems Group of IBM Research. The NASSL technical report states that “Languages such as MS IDL and OMG IDL are popular industry standards used to describe RPC interfaces. Instead of inventing yet another language, NASSL borrows key concepts from some of these languages and adapts them to cover non-RPC types of services”. Examples of concepts taken into NASSL and WSDL from the OMG’s and Microsoft’s Interface definition languages include oneway operations, the failure semantics and strong typing. We will further determine the origins of Microsoft IDL and OMG IDL in Sections 7 and 8. The most significant original contribution of the NASSL technical report itself is the proposed use of different type systems, such as XML schemas, for the definition of parameter and return types of web services, which eventually was adopted for WSDL.

The SOAP Contract Language (SCL) and the Service Description Language (SDL)3 were defined by Microsoft in 1999 and 2000 to support a web service based remoting model for Microsoft’s Component Object Model. SDL was relatively widely used and the definition of SDL was distributed as part of the Microsoft SOAP toolkit. The definition of SCL was relatively short-lived and SCL was superseded by WSDL relatively quickly. The main influence of Microsoft’s SDL on WSDL and modern web services was the ability to use different bindings to underlying transport protocols. The SDL document gives examples of HTTP, HTTPS and Microsoft’s RPCs as used in DCOM.

WSDL, NASSL, SDL and SCL are all XML languages. The first proposal to use XML to express an interface definition language for web services was made by [Merrick and Allen 1997] in a Note published by the World-Wide-Web Consortium proposing a Web Interface Definition Language (WIDL). Although we are unable to present direct evidence that WSDL, NASSL, SDL and SCL did indeed borrow this idea from WIDL this seems highly likely for a number of reasons. Firstly, WSDL, NASSL, SDL and SCL were defined while Microsoft and IBM were very much involved in the Protocol Working Group of the World-Wide-Web Consortium. Secondly, WIDL was submitted as a Note in 1997 to the World-Wide-Web Consortium. Finally and most importantly, the XML Protocol Working Group that eventually adopted WSDL published a matrix that compared the different approaches to defining web services in 2000 [Prudhommeaux 2000]. The matrix compares amongst others WIDL, WSDL, NASSL and SCL, which means that the language designers of WSDL were aware of the WIDL approach.

3Note that even though the same acronym is used for all of them, Microsoft’s Service Description Language is fundamentally different from the ISO/IEC Specification and Description Language and should also not be confused with Microsoft’s own Security Development Lifecycle.
4.4 Business Process Execution Language

[Peltz 2003] identifies two models of composing Web services: orchestration and choreography. The primary difference lies in the scope of composition. While orchestrated web services consider control flow from one party’s view, choreography emphasizes collaboration of each party and encompass the scope of all parties and their interactions on the global view of a system. BPEL is intended to orchestrate Web services along with a business process. A diagram showing the impact traces that led to BPEL is shown in Figure 8.

There are a significant number of BPEL implementations. IBM developed BPWS4J (Business Process Execution Language for Web Services Java Run Time). Microsoft BizTalk Server 2004 translates WS-BPEL into XLANG/s then .NET assemblies for better performance while keeping backward compatibility to previous BizTalk servers [Fancey 2005]. Oracle bought Collaxa and developed Collaxa’s BPEL suite into Oracle’s BPEL Process Manager which can run both standalone and an option to Oracle Application Server Enterprise Edition. There are a number of open-source BPEL engines available from Active Endpoints, RedHat/JBoss and Apache. Graphical modelling support for BPEL is available under an open source license through the Eclipse BPEL Designer project jointly supported by IBM, Oracle and UCL.
The first specification of BPEL named BPEL4WS (Business Process Execution Language for Web Services), was published by IBM and Microsoft in August 2002. BPEL4WS states that it consolidates two earlier languages developed by IBM and Microsoft respectively. These are IBM’s Web Services Flow Language (WSFL) [Leymann 2001] and Microsoft’s XLANG [Thatte 2001] used in Microsoft’s Biz Talk Server product family.

WSFL emerged from IBM’s earlier work on FlowMark [Leymann and Roller 1994; Leymann and Altenhuber 1994]. [Alonso et al. 1995] of IBM Almaden claim compliance of FlowMark to the WfMC standard when they state “FlowMark follows very closely the reference model provided by the WfMC”. Further evidence that the FlowMark team were fully aware of the Workflow Management Coalition (WfMC) is the fact that Altenhuber was a technical committee member of the WfMC.

One of the key artifacts of the WfMC is its Reference Model [Hollingsworth 1995]. The model does not include any references but states that, amongst others, the reference model evolved from “Project Support Software”. It goes on to state: “Software to handle complex IT application project development (eg IPSEs - “Integrated Project Support Environments”) has often provided a form of workflow functionality within the project environment, for “transferring” development tasks between individuals and routing information between individuals to support these tasks”. Were the reference model written as a scholarly publication it would have referenced the extensive literature on process-centred software development environments, such as [Barghouti 1992].

[Meredith and Bjørg 2003] propose the formal definition of behavioral aspects in BPEL using the Polyadic π-Calculus [Milner 1999]. According to [Leymann et al. 2002] this approach was used already in XLANG: “Our use of directed graphs to model flows is similar to the work within the Workflow Management Coalition (WfMC) or the Web Services Conversation Language. The theoretical underpinning of this approach is rooted in Petri Nets and the π-calculus. Languages closer to the latter are XLANG and BPML.”

4.5 Summary

The three key web service technologies that enable the construction of service oriented architectures are SOAP, WSDL and BPEL. We have shown that they are widely implemented in service oriented middleware systems. These web services technologies use XML technologies to define message content of service invocations, interfaces to web services and to determine web service compositions. We have traced the origins of XML technologies to the GML effort at the IBM Cambridge lab based at MIT. We have shown that the key concepts used in web services are not new but are rather taken from remote procedure call, distributed object and workflow systems.

Service-oriented architectures are often layered onto distributed component systems, such as the Java 2 Enterprise Edition. These component systems provide containers for components that are typically only distributed on a local area network. These containers provide the mechanisms for persistence of component state, replication, concurrency control, component activation and deactivation, security and transaction management. In the next section we discuss the origins of distributed transaction processing in such distributed component systems.
5. DISTRIBUTED TRANSACTION
5.1 Distributed Transactions in J2EE

Java application servers commonly implement the Java 2 Enterprise Edition (J2EE) defined by Sun Microsystems. It consists of a number of constituent specifications. These include Java Server Pages (JSP) and Servlets, Java Messaging System (JMS) and Enterprise Java Beans (EJB). The EJB specification [DeMichiel and Keith 2005] is at the very core of J2EE as it provides a server-side component model, which is capable of providing persistence, replication, transactions, security and concurrency control services in a way that their use can be kept transparent to application programmers.

In this section, we investigate a particular aspect of the origins of this EJB technology. It would be impossible to provide a full account, but we rather endeavour to highlight some of the key ideas that shaped the EJB specification. We deliberately ignore the impact that research has had on the definition of the Java programming language as this aspect is covered in a companion impact report by [Ryder et al. 2005]. We now investigate distributed transaction processing in the J2EE standard. This is of particular significance as J2EE application servers, would not be very useful if it was not possible for multiple users to interact with applications at the same time, or if failures that are bound to occur could possibly corrupt data held in various distributed data stores. Before investigating the origins of distributed transactions in J2EE we characterise the economic significance of the J2EE application server market.

The J2EE application server market had a size of USD 2.19 billion license revenue in 2001 according to [Gilpin 2002] of Forrester Research. Figure 9 shows the distribution of this market across product vendors. The two main products in the market at the time were the IBM Websphere J2EE application server and WebLogicServer of BEA Systems, which each had a share of 34% of the market. Further products include Sun’s iPlanet, Oracle 9i, as well as the Sybase and HP Arjuna application servers. We note that JBoss [Fleury and Reverbel 2003] is widely used and freely available J2EE application whose impact is not reflected here.

![Figure 9. J2EE application server market in 2001](image)

Figure 10 shows an overview of the time-line and key impact traces concerned with distributed transaction management in J2EE. Vendors of J2EE application servers implicitly
acknowledge the fact that they use the J2EE specifications by virtue of claiming compliance. For the purpose of transaction management, there are two significant J2EE specifications that a J2EE application server needs to comply with, the Java Transaction Service (JTS) specification [Cheung 1999] and the Java Transaction API specification [Cheung and Matena 1999]. The EJB specification determines two models for transaction management. The first model is container managed transactions where the container that manages the life cycle of EJB components determines when to start and when to complete transactions. The second approach allows the application programmer to determine the boundary of transactions and to do so they use the interfaces defined in the JTA specification. Both the container and the implementation of the JTA that an application programmer uses, in turn, rely on a JTS implementation to correctly implement transactions.

The introduction of the JTS specification states that “JTS uses the CORBA Object Transaction Service (OTS) interfaces for interoperability and portability [...]. These interfaces define a standard mechanism for any implementation that utilizes IIOP (Internet InterORB Protocol) to generate and propagate transaction context between JTS Transaction Managers. Note, this also permits the use of other API over the IIOP transport mechanism to be used; for example, RMI over IIOP is allowed.” [Cheung 1999]. The JTS specification therefore merely defines a Java programming language binding to a subset of the CORBA Object Transaction Service and thus it is the origins of the CORBA OTS that are of particular interest for the remainder of this section.

5.2 CORBA Object Transactions

The main aim of the OTS was to encapsulate the protocols required for distributed transaction processing in object-oriented interfaces while preserving the interoperability between different transaction monitors and databases achieved by earlier standards.

We cannot consider the OTS in isolation but also have to take the CORBA Concurrency Control Service (CCS) into account, as the OTS uses the CCS to achieve isolation of transactions. Both these services were defined as part of the second call for CORBA Services proposals by the OMG membership in 1993-94. A collection of all CORBA services that includes the OTS and CCS is published in [OMG 1998a].

Five consortia responded with a proposal for the OTS. A quite influential consortium consisted of IBM, Transarc and Tandem and was supported by Hewlett Packard, Objectivity and NCR. Their proposal [Object Management Group 1993b] was co-authored by Edward Cobb of IBM, Ken McCoy of Tandem and Graeme Dixon of Transarc. A second influential proposal [Object Management Group 1993a] was co-authored by Herve Lejeune of Bull, Chris Horn of Iona and Juan Andrade of Novell. By virtue of having leading transaction monitor products (CICS, Pathway, Tuxedo and Encina respectively), IBM, Tandem, Novell and Transarc had a vested interest in the Object Transaction Service. The final specification was submitted by merging key contributions from all competing proposals and was created by a consortium that included IBM, Transarc, Tandem, Novell, Iona, ICL, Bull and Sun. A good overview of the final object transaction service is provided by [Cobb 1997].

The IBM/Tandem/Transarc proposal includes the key abstractions of transactional client, server and manager that were eventually incorporated into the standard. It also defined the notion of implicit transactions (those that are not explicitly identified through a transaction ID), which is a key component of the OTS standard. A key contribution of the Novell proposal was the notion of resource managers and the interoperability with the XA proto-
Fig. 10. Impact traces for distributed transactions
on the transaction processing architecture of BEA’s WebLogic Server.

The object transaction service relies on a concurrency control service for achieving the isolation property. Two concurrency control service proposals were submitted. The first was also written by Graeme Dixon of Transarc and the second proposal was submitted by Emeraude and supported by members of the PCTE consortium (ICL, Bull, Siemens). At their meeting in Dublin in August 1994 the OMG Technical Committee voted unanimously in favour of adoption of Dixon’s proposal as it was integrated with the object transaction service and provided a truly object-oriented view on concurrency control, while the PCTE submission merely proposed adoption of the PCTE concurrency mechanisms.

So where did Dixon, Cobb and Andrade get the ideas embedded in the concurrency control and object transaction service specifications? Figure 11 shows a refinement of Figure 10 in order to highlight in more detail the origins of the CORBA OTS proposal.

At the time of specifying the OTS, Andrade worked at Novell in the department that developed the Tuxedo transaction monitor. Tuxedo was later acquired by BEA from Novell and the transaction processing experience gained by Tuxedo forms a cornerstone upon which BEA’s Web Logic Server product is built. Andrade also worked on the OSF ODTP standard and determined the XA protocol. Novell had earlier obtained Tuxedo from AT&T Bell Labs where Andrade et al had built an experimental transaction monitor known as System/T.
Both the CCS and OTS specifications used CORBA IDL as established in [Sventek and Andreas 1991] in order to define the interfaces to their respective services. The concurrency control service specification explicitly cites [Bernstein et al. 1987] when describing two-phase locking. It also refers to [Gray 1978] for the notion of a transaction. Gray cites in this seminal paper [Lampson and Sturgis 1976] as the first mention of the two-phase commit protocol. Note that contrary what Gray states, the paper was never published in CACM, but was circulated amongst systems researchers in different versions and eventually appeared as a XEROX technical report in [Lampson and Sturgis 1979]. Encina, Transarc’s TP Monitor also supported the execution of nested transactions and Dixon referred to [Moss 1981; 1985] in the concurrency control service specification in order to ensure support for nested transactions was included in the specification.

The Novell OTS proposal includes an extensive list of citations. It defines a CORBA OTS transaction to have the meaning of Gray’s ACID transactions although it cites [Gray 1993] rather than his original LNCS paper. The Novell proposal also refers to the X/Open DTP specifications and states that resource managers can communicate via the XA protocol [X/Open Group 1991] with data stores. The X/Open DTP XA specification is ultimately just an API specifications. The underlying protocols and services were specified in the ISO 10026 standard, which is explicitly referenced, as well as the underlying two-phase commit and two-phase locking protocols as defined in the ISO 9804 and ISO 9805 standards. Like the CCS proposal, the Novell OTS proposal supported the notion of nested transactions and again, they acknowledge [Moss 1981] as the origin.

At the time of writing the OTS proposal, Dixon worked with Michael W. Young as chief architect for the Encina product of Transarc. Transarc was spun out of Carnegie Mellon University by Alfred Spector to develop distributed transaction processing products. A paper that describes the architecture of Encina was published by [Young et al. 1991]. Daniel Thompson, one of the co-authors of that paper had previously worked with Spector on the Camelot project at CMU [Spector et al. 1986] and Camelot was extending ideas for distributed transaction processing that were first built in the TABS system [Spector et al. 1985]. Encina used many concepts that were built into Camelot and TABS, including nested transactions.

Transarc was acquired by IBM in 1994 and the Encina transaction engine became an integral part of the IBM WebSphere Application Server from Version 3.0. Transarc also licensed the Encina Transaction engine to Iona, who marketed it as the Iona Object Transaction Monitor and to Netscape and Sybase.

While basic distributed transaction handling and nesting of transactions were contributed by members of the Camelot team, it was Dixon who contributed object-oriented abstractions to distributed transaction processing. Dixon obtained his PhD in the distributed systems group at Newcastle University, where he was a key contributor to the Arjuna project [Shrivastava et al. 1991]. His research was on concurrency control and transactions in persistent distributed object systems [Dixon et al. 1989]. In his PhD thesis [Dixon 1988], he investigated how distributed transaction processing could be facilitated using an object-oriented approach and the key concepts that underpin the CORBA Object Transaction and Concurrency Control Services were first defined during his PhD work in Newcastle. These include the notion of server objects that could take part in atomic actions as well as the need for integration of object transaction services with concurrency control and persistence. Thus, contrary to the claim in the IBM Patent it was the Arjuna project that
had the first object-oriented implementation of a distributed transaction capability between 1985 and 1990.

Arjuna used Rajdoot [Panzieri and Shrivastava 1988], an RPC mechanism supporting orphan detection techniques to detect transactional clients and servers that had failed. This paper cites [Birrell and Nelson 1984] in order to acknowledge that Birrell had the first ideas for RPCs. It is interesting to note that, once adopted, the OTS and JTS specifications had reciprocal influence on the Arjuna project. The details of this influence have been documented by [Little and Shrivastava 2002] in more detail than we are able to give in this report. In essence, the Arjuna project tracked the standards development in their work and released versions of the Arjuna middleware that were compliant to the OTS and JTS specifications respectively.

They then transferred the IPR of the middleware into a spin-off company, Arjuna Labs. Arjuna Labs were then bought by BlueStone, who were, in turn, acquired by Hewlett Packard and the Arjuna JTS implementation now provides the transaction engine of HP’s application server product. Recently, the Arjuna transaction service technology was sold to JBoss, who were in turn acquired by RedHat and the Arjuna JTS is now being integrated into RedHat’s Linux offering.

5.3 Summary
We have shown the importance that research has played in the definition of distributed transaction processing capabilities that form one of the most important foundations of J2EE and CORBA specifications and products. We have shown that basic research results published in the software engineering and database literature and PhD theses have had a key impact in this important area.

Distributed Transactions also play a key role in another cornerstone of the J2EE specification, the Java Messaging Service. It is the distributed transaction protocols we have discussed above that enable components to enqueue or receive a distributed message as part of a transaction and most JMS message queue implementations are transactional resource managers and implement the XA protocol we have discussed above. We investigate the origins of JMS and more generally message-oriented middleware in the next section.

6. MESSAGE-ORIENTED MIDDLEWARE
Message-oriented middleware supports the asynchronous communication between distributed system components. This is a very important primitive used in most modern application servers and enterprise services buses. The Java 2 Enterprise Edition includes a specification called the Java Messaging Service (JMS) [Hapner et al. 2001]. JMS is well integrated with other J2EE specifications. In particular, the Enterprise Java Bean specifications from Version 1.2 onwards define the notion of message beans, which can be configured to be invoked when a message is received via the Java Messaging Service.

According to [Correira and Biscotti 2006], the size of the market for MOM licenses was about 1 billion US$ in 2005, with the biggest vendors of message-oriented middleware being IBM with a market share of about 500 million US Dollars and TIBCO with a market share of about 314 million US Dollars. It thus reflects a non-negligible proportion of the overall middleware market. At the time of writing this article, the most significant products are IBM’s Websphere MQ, the BEA MessageQ, Tibco’s TIB/Rendezvous, Sonic’s SonicMQ, Microsoft’s MSMQ, as well as a number of open source products, such as the JBoss Message Queue. These products are not just sold in isolation but are also bundled...
into application servers and enterprise service buses.

An important focal point for the adoption of message-oriented middleware market was the publication of the JMS specification as the result of a Java Community Process. The aim of this specification is to provide a uniform application programming interface for all leading message-oriented middleware providers. JMS supports both point-to-point and publish/subscribe communication. It supports the definition of different qualities of service, such as guaranteed delivery through persistent storage of messages if receivers are unavailable. These concepts were not invented by the committee that defined JMS but have been used by message-oriented middleware products for a number of years before. A survey of different message-oriented middleware approaches, including those that have not yet had any significant commercial success is provided by [Eugster et al. 2003]. In the remainder of this section, we investigate the origins of key concepts of commercially successful message-oriented middleware systems. An overview of the key impact traces in the area of message-oriented middleware is shown in Figure 12.

Fig. 12. Impact on JMS implementations

The message-oriented middleware product with the largest market share is IBM’s WebSphere MQ, which was previously referred to as MQSeries [IBM 1995]. MQSeries was successful for a number of reasons. Firstly, it integrated messaging primitives with transaction processing by making the messaging middleware compliant to the OpenGroup’s XA protocol that is part of the Open Distributed Transaction Processing standard. The treatment of a message queue as an XA compliant resource manager allowed programmers to insert or remove a message into or from a queue as part of a distributed transaction that would also perform other database operations to process sending or receiving the message. Secondly, MQSeries was made available for more than 30 operating systems and hardware.
platforms and allowed the reliable exchange of messages using the same primitives across these platforms.

The creation of MQSeries as a point-to-point message queuing system was facilitated by State Street Bank in Boston, which demanded the integration of the EzBridge product of System Strategies International (SSI) with the MQI abstractions that were defined as part of IBM’s Networking Blueprint [Hess et al. 1995]. To meet this request, IBM UK Laboratories in Hursley where the CICS product originated licensed the ezBridge Software for Tandem and VMS, provided it with MQI interfaces, made it XA compliant and integrated it with its own MQI implementation that was available for the CICS platform. We interviewed Dick Dievendorff, the technical lead of MQSeries on MVS. Dievendorff stated that the MQSeries team used the ideas of forward recovery to overcome the uncertainty in the two-phase commit protocol. He mentioned that his team used the forward recovery techniques published in the 1989 VLDB paper by [Rothermel and Mohan 1989]. In fact, this paper won the VLDB most influential paper award in 1999 and [Mohan 1999] summarizes the history of ARIES retrospectively. On the relationship between ARIES and MQSeries this paper states: “Asynchronous program to program communication in the form of transaction-based persistent messaging has become quite popular. IBM’s MQSeries, for example, is a very successful product in this arena. MQSeries includes its own non-DBMS-based persistent storage mechanism, using an extended version of ARIES, for managing the messages” [Mohan and Dievendorff 1994].

In later correspondence about an earlier draft of this article, Dievendorff also confirmed that the design of the concurrency control mechanism in MQSeries used the seminal work of Jim Gray on transaction management concepts and in particular the locking mechanisms Gray proposed. The team also used the source code of the System R lock manager [Astrahan et al. 1976].

In an email correspondence Rob Drew, who was an architect for MQSeries at the time, confirmed that the initiative for developing MQSeries might have been influenced by the availability of an earlier competing product, the DEC Message Q. As it turns out, the DEC Message Q turned out to be quite influential, too.

In a June 2005 interview with Paul Krill of InfoWorld, Paul Patrick of BEA confirmed that BEA acquired the ObjectBroker (a CORBA ORB) and the MessageQueue products from DEC and re-engineered the ObjectBroker into their Web Logic Enterprise (WLE) CORBA implementation. WLE implemented the transaction service based on Tuxedo discussed in the previous section. WLE implemented the CORBA Notification [OMG 1998b] by wrapping the BEA MessageQ. An example of an application that was built using WLE’s transaction and notification services is described in [Emmerich et al. 2001]. Before incorporating MessageQ into WLE, BEA marketed the MessageQ separately and also bundled it into their Tuxedo transaction monitor.

So where did the messaging primitives DEC built into their message queue originate? DEC had earlier developed a product called DEC Fuse. The aim of DEC Fuse was to use asynchronous and anonymous communication techniques to support the integration of software engineering tools. Fuse was based on code that DEC licensed from Steven Reiss, who had developed a message-passing based integration mechanism for the Field software development environment [Reiss 1990]. [Hart and Lupton 1995] state in their DEC Journal paper that describes Fuse that “code that determines Fuse was based on Field Code”. In fact also Sun’s ToolTalk and Hewlett Packard’s Broadcast Message Server tool integration
products were based on Field’s concepts. In his HP Journal paper on SoftBench, [Cagan 1990] states that “Most influential in the HP Softbench design were the FIELD system done at Brown University, the FSD system done at USC-ISI and the HP NewWave environment”. When referencing Field, Cagan refers to a 1987 Brown technical report written by [Reiss 1987] and for the FSD system he cites [Balzer 1987]. When interviewing Steven Reiss, he confirmed that HP were made aware of Field during a visit at Brown and that he licensed the code of Field for inclusion in Fuse. Moreover he confirmed that Fuse was a predecessor of the DEC Message Queue.

It is interesting to note that the CORBA Event Service [OMG 1998a], an important predecessor of the Notification service was defined by a consortium that was led by Sun, DEC and HP. It is unclear though which influence, if any, Sun’s, DEC’s and HP’s previous work on the Tooltalk, Fuse and Broadcast Message Server tool integration mechanisms has had on these developments. Also Dale Skeen of Teknekron took part in the task force that defined the CORBA Event service. Skeen had worked previously at Teknekron on a product called The Information Broker [Skeen 1992]. In fact, the paper published by [Oki et al. 1993] in the 1993 ACM SIGOPS Principles of Operating Systems Conference describes a publish/subscribe mechanism that is very similar to that used by the CORBA Event service. In the discussion of related work Oki et al. state that they apply the ideas of publish/subscribe to objects that were proposed by [Cheriton and Deering 1988] for multicast networking and by [Birman and Thomas 1989] for building the ISIS replication manager. Following the acquisition of Teknekron by Reuters, the company was renamed into Tibco and The Information Bus is now merely known by its acronym TIB, which has the second largest share of the message-oriented middleware market.

While message queuing middleware is frequently used to provide primitives for asynchronous communication between distributed system components, they are less suitable for synchronous communication. These are typically provided using remote method invocation and remote procedure call primitives. We discuss the origins of middleware that provide such primitives in the next two sections.

7. DISTRIBUTED OBJECTS

For the discussion in this section we consider the techniques that support the invocations of operations amongst objects between machine boundaries. The scope therefore includes object requests as supported by the Common Object Request Broker Architecture, the Remote Method Invocation primitives of Java and the Remoting architecture of .NET.

For developing modern distributed applications, software engineers use remote invocation primitives that are now tightly integrated into programming languages. We have already discussed the size of the J2EE application server market above. The Enterprise Java Bean specification [DeMichiel and Keith 2005] that is an integral specification to which a J2EE application server needs to comply prescribes the use of Java’s Remote Method Invocation (RMI) [Sun Microsystems 1998]. The specification determines that ‘The remote and remote home interfaces are Java RMI interfaces’ so that EJBs can be accessed across machine boundaries. .NET uses a primitive called Remoting [Srinivasan 2001] for that purpose. Java RMI and .NET Remoting therefore provide the basis for invoking operations between different distributed tiers of a software architecture without which scalable and reliable web applications could not be built.

[Alur et al. 2003] and [Microsoft 2003] define the best practices for distributed software
design with the Java 2 Enterprise Edition and .NET, respectively. Given they originate at Sun and Microsoft respectively, these best practices are widely followed in the software industry and a substantial number of e-commerce, online banking and other web sites are developed using these practices. [Alur et al. 2003] propose a strict separation between a presentation layer and business object layer. The presentation layer is in charge of computing the web content that gives the user interface to an application. The business object layer is concerned with handling the application logic and interfacing to the underlying information systems and legacy applications. It is best practice to host these on different machines so as to support load balancing and fault tolerance. As a consequence objects in the presentation layer need to invoke operations from objects in the business object layer across machine boundaries. This is achieved in practice using Java Remote Method Invocation or .NET Remoting.

We can therefore establish that remote operation invocation primitives are of crucial importance for the construction of distributed software applications and we can now start investigating where these remote invocation primitives originated.

7.1 Java Remote Method Invocation

The Remote Method Invocation definition at Sun Microsystems [Sun Microsystems 1998] was led by Jim Waldo and its technical concepts were first published in [Wollrath et al. 1996]. RMI enables a Java client object to invoke a method from a server object that resides in a different JVM, possibly on a different host.

Figure 13 shows the impact traces that led to RMI. [Waldo 1998] describes the main design rationales and origins of RMI. Waldo acknowledges that RMI builds on the heritage of remote procedure call systems and identifies [Birrell and Nelson 1984] as the first RPC system. He also cites the OMG’s Common Object Request Broker Architecture [Object Management Group 1991] and Microsoft’s Distributed Component Object Model [Grimes 1997] as precursors of RMI. Waldo then outlines the design rationales for RMI and states important differences to earlier work on RPCs, CORBA and DCOM. In particular, RMI gives up the notion of programming language heterogeneity that is resolved by an independent interface definition language and different programming language bindings that underpinned all earlier work on remote procedure call systems. Instead [Wollrath et al. 1996] state that RMI continues the tradition of Modula-3 Network Objects [Birrell et al. 1993] that incorporate remote invocation primitives into the programming language. [Wollrath et al. 1996] also states that RMI builds on the garbage collection mechanism first adopted in Modula-3 Network Objects.

RMI also differs from previous work on remote procedure calls in that the stubs required for synchronisation, marshalling and unmarshalling are loaded on-demand rather than during startup of the distributed program. This mechanism considerably simplifies the deployment of distributed programs. To achieve loading of stubs on demand, RMI depends on Java’s dynamic class loading mechanism described by [Liang and Bracha 1998]. For use in distributed computing, the security of class loading is critical and [Liang and Bracha 1998] describe how class loading is made both type safe and secure following the formal proof of flaws of earlier Java class loading mechanisms by [Jensen et al. 1998].

[Birrell et al. 1993] begin their Network Objects paper by stating that “in pure object-oriented programming, clients cannot access the concrete state of an object directly, but only via the object’s methods. This methodology applies beautifully to distributed computing, since the method calls are a convenient place to insert the communication required by
the distributed system.” Thus, Network Objects continue in Wirth’s and Parnas’ tradition of encapsulation, modularity and information hiding and build on the plethora of work on object-oriented programming languages. We refer the interested reader to [Ryder et al. 2005] for an in-depth discussion of the origins of this work.

[Birrell et al. 1993] state explicitly that Modula-3 Network Objects are built on ideas of others and reference Emerald [Jul et al. 1988], Argus [Liskov 1988], Eden [Almes et al. 1985], Arjuna [Dixon et al. 1989], Orca [Bal et al. 1992] and SOS [Shapiro et al. 1989] as systems that they built upon. Many of these papers are the journal publications of technical reports or theses describing systems in more detail. The object model of Emerald is defined in [Hutchinson 1987] and its type system was described in [Black et al. 1987], the Arjuna persistent object approach was built by [Dixon 1988] and the Orca object model was defined in [Bal 1989].

Birrell et al. claim three main contributions in their paper. The first one is that Net-
work Objects are considerably simpler and more lightweight than other approaches; they achieve this by selecting only those concepts from earlier systems they deem essential for distributed programming. Their second contribution is the integration into a coherent language framework and their third contribution is an implementation of that language that they have then subjected to rigorous quantitative and qualitative evaluation in order to show the feasibility of relatively fine grained distribution. The research into Modula-3 Network Objects in DEC’s industrial research lab can thus be considered as a necessary and, given the impact it had on RMI, very influential consolidation step of earlier, largely academic work.

Without doubt, Network Objects were also influenced by Andrew Birrell’s earlier work on remote procedure calls [Birrell and Nelson 1984], but before discussing this we would like to trace the impact of research on another direct predecessor of RMI, which is CORBA as acknowledged in [Waldo 1998]. As a matter of fact, Jim Waldo was a member of the team that defined the first CORBA specification when he was working at HP/Apollo. RMI more or less directly reused a number of key concepts from CORBA, including the use of exceptions to handle communication failures, location transparency of objects through remote object references, interface inheritance with a well defined root object type, the use of stubs for marshalling and unmarshalling, the default synchronisation behaviour, the ability to perform on-demand activation, and the Internet Inter ORB Protocol (IIOP) as a transport representation.

7.2 The Common Object Request Broker Architecture

The CORBA standard went through a significant number of revisions between 1991 and 2003. With the notable exception of the IIOP protocol, which was added in CORBA 2.0 [Object Management Group 1995] all the above concepts that made their way into RMI were already included in the very first CORBA version and we therefore have to investigate the origins of that specification.

The impact trace for CORBA is shown in Figure 14. The Common Object Request Broker Architecture was specified in an open process by the Object Management Group. The first CORBA specification is the result of two task forces, the first of which defined the Object Model and the second determined the Object Request Broker Architecture.

One can identify two main areas of influence from different research disciplines on the first CORBA specification. Research on software engineering and programming languages has influenced the CORBA object model and research on distributed systems has influenced the ORB architecture. We discuss both of these areas below.

It is evident that the team that defined the object model was fully aware of, and building on, previous research and development efforts in object oriented programming languages, object-oriented design and object databases. The OMG Technical Committee meetings during that period had regular presentations from leading researchers on a topic of relevance to CORBA. These presentations included talks from Bjarne Stroustrup on C++ and the need for co-existence and integration of C++ with other languages (he mentioned Smalltalk in particular) at the TC Meeting in February 1990, Klaus Dittrich on Database Object Models at the TC Meeting in April 1990, Ivar Jacobson on Objectory at the TC Meeting in June 1990 and Andrew Watson on ANSAware at the TC Meeting in October 1990. Moreover, the document that eventually defined the object model [Soley 1992] has an extensive list of references, which includes the object database manifesto [Atkinson et al. 1990], references to object-oriented programming texts, such as [Meyer 1988],
Fig. 14. Impact traces leading to OMG/CORBA

[Keene 1989] and [Cox 1986] and extensive list of references to object-oriented design books, including [Booch 1991] and [Wirfs-Brook et al. 1990]. The object model was determined in a collaborative manner by one of the very first OMG task forces as part of an effort to write a ‘standards manual’. This manual was edited by Richard Soley, adopted in its first version in September 1990 and eventually published as a book by John Wiley [Soley 1992]. The core object model and the glossary were contributed by [Snyder 1990b; 1990a], who was at Hewlett Packard Labs at the time. Both are clearly influenced by Snyder’s earlier research work. The notion of encapsulation and the treatment of multiple inheritance in the core object model bear considerable resemblance to the concepts outlined in [Snyder 1986]. The object model enforces data abstraction in that the core object model does not allow any direct access to data, but attributes are rather treated as (pairs of) accessor operations, which is very much in the tradition of data abstraction in CLU [Liskov et al. 1977]. The exception handling that the CORBA object model profile adds to the core object model is strongly influenced by Snyder’s earlier work on exception management in CLU [Liskov and Snyder 1979].

crosystems in January 1991. In February 1991, Software AG, Teknekron and Constellation withdrew their submission. Sun dropped its proposal and joined the HP submission [Solley 1991]. The ANSA submission was edited by John Bull, Andrew Herbert and Andrew Watson, with contributions from other ANSA team members. The Advanced Network Systems Architecture Reference Manual [ANSA 1989] was produced between 1984 and 1989 by a team led by Andrew Herbert and including, Joe Sventek, Dave Otway, Owen Rees and others. Sventek left to join HP in 1990, where he co-authored the HP ORB proposal. ANSA continued to influence OMG work through the mid-90s, not least because Andrew Watson of ANSA chaired the OMG ORB Task Force. The HP/Sun resubmission was written by and presented to the OMG TC in March 1991 by Joe Sventek, who led the ANSA implementation before joining HP, and Jim Waldo of Hewlett Packard and Richard Probst and Dwight Hare of Sun. At the OMG TC Meeting in May 1991, APM withdrew their submission, stating that 'APM would be unable to handle the burden should their submission be successful, that APM is a research consortium that would prefer to work on "future" ORBs and ODP and that the ORB process is well supported by Hewlett Packard and Digital Equipment Corporation'. Following an independent review of the remaining submissions by [Shapiro 1991] of INRIA it was also decided at that May meeting to reject all submissions and invite the remaining submitters to prepare a joint submission. This joint revision was edited by Joe Sventek and Bill Andreas of Hyperdesk and submitted in August 1991 on behalf of HP, DEC, HyperDesk, NCR and Sun [Sventek and Andreas 1991]. Sventek and Andreas presented the proposal to the OMG Technical Committee Meeting in September 1991 and it was subsequently adopted as a standard. Andrew Watson chaired the ORB Task Force from 1992 while working for APM; in 1995 he left to join OMG as Architecture Director, and later Technical Director.

Thus we have presented evidence that the research on ANSAware at APM to which Herbert, Watson and Sventek were key contributors and which was funded in part by the EU ESPRIT programme, had a very significant influence on the CORBA architecture. In the next section we show the relationship between ANSAware and RPC systems.

8. REMOTE PROCEDURE CALLS

To characterise the importance of remote procedure calls we cannot use market size as an indicator as these primitives have been commoditised and no longer have a distinct market. Instead remote procedure call primitives are embedded into all major operating systems. Every installation of Windows NT, Windows 2000, Windows XP, every Unix installation, including Linux and all Apple Macintosh computers running Mac OS X routinely use remote procedure calls for communication with other computers. Remote procedure call primitives are used by modern operating system itself to perform very basic functions. Examples that use RPCs are the management of the Domain Name Service [Mockapetris and Dunlap 1988] and the management of the DHCP service that assigns IP addresses dynamically, which implements logical Internet domain names. We can therefore postulate that we would not have the Internet in its current form without remote procedure calls.

There have been a large number of remote procedure call mechanisms proposed. These included Xerox Courier [Xerox OPD 1981], Xerox Cedar [Birrell and Nelson 1984], Apollo NCA [Dineen et al. 1987], Sun ONC/RPC [Sun Microsystems 1987; 1988], Cambridge Mayflower RPC [Bacon and Hamilton 1988], MIT Athena RPCs [Souza and Miller 1986], Stanford Modula/V RPCs [Cheriton 1984; Almes 1986], ANSAware [ANSA 1989]
and Rajdoot RPCs [Panzieri and Shrivastava 1988]. A comprehensive survey that presents and compares these approaches is provided by [Tay and Ananda 1990]. With hindsight, we are now able to say that there are three strands of remote procedure calls that had significant impact. An overview of the relevant impact traces is shown in Figure 15 below.

![Impact traces for remote procedure call systems](image)

The first strand is ANSAware, which influenced the ODP standard and CORBA as described above. The ANSA Reference Manual [ANSA 1989] states that “the protocol for remote operation invocation is very similar to remote procedure calls” and explicitly cites the Cedar RPC mechanism of Xerox [Birrell and Nelson 1984]. ANSAware includes an interface definition language (IDL), a compiler for that interface definition language that generates stubs, which resolve data heterogeneity and implements presentation and marshalling in very much the same way as proposed by [Birrell and Nelson 1984].
The second strand are RPCs as defined by Sun’s Open Network Computing [Sun Microsystems 1987; 1988]. ONC has been highly influential as it defines the RPC mechanism that is built into any Linux, Mac OS X, Solaris and other BSD Unix variants. Evidence for this influence is the fact that the RPC manual pages on these Unix variants explicit reference the relevant IETF RFCs that define ONC RPCs. Amongst these, IETF RFC 1050 [Sun Microsystems 1988] refers to Andrew Birrell’s and Bruce Nelson’s work by stating: “This document specifies a message protocol used in implementing Sun’s Remote Procedure Call (RPC) package. [...] It does not attempt to justify RPC or its uses. The paper by Birrell and Nelson [1] is recommended as an excellent background to and justification of RPC.” Thus, the IETF RFC explicitly references the XEROX technical report that was eventually published as [Birrell and Nelson 1984] as providing the background and rationale for using RPCs.

The third strand is the NCA RPC strand that heavily influenced OSF/DCE RPCs and Microsoft’s RPC. The “What is RPC?” page of Microsoft’s TechNet [TechNet 2003] lists Microsoft products and technologies that depend on Microsoft’s RPC implementation, which includes DCOM, COM+, the management interfaces to DHCP and DNS services, the Distributed Transaction Coordinator and many others. The relationship between Microsoft RPCs and DCOM was confirmed by Paul Leach in an interview when he said that “DCOM is just a slight extension of MS RPC”. Leach also confirmed that Microsoft were fully aware of Birrell and Nelson’s paper when they defined MS RPCs. Microsoft’s RPC implementation, however, evolved from the Distributed Computing Environment (DCE) of the Open Software Foundation (OSF). A comprehensive account on the relationship of Microsoft’s RPCs and DCE/RPCs is given in Chapter 7 of [Shirley and Rosenberry 1995]; we can assume this reference to be authoritative as Rosenberry was also an editor of the OSF/DCE specifications.

OSF/DCE specifies a number of technologies and its remote procedure calls are defined in [Open Group 1997]. DCE was defined by the Open Software Foundation, a precursor to the OpenGroup in a public standardization process. DCE comprises a number of basic technologies that are required to build secure distributed systems. These technologies include a remote procedure call facility, distributed file system, a distributed time service, directory services and a standardized threading library. Unfortunately, the DCE standards documents themselves do not include any references to the origins of the technology that was adopted. However, several sources state that the RPC mechanism of DCE emerged from Apollo’s Network Computing Architecture (NCA) that was incorporated into HP-UX following Apollo’s merger with Hewlett Packard:

—Kong 1995] states in his HP Journal article on DCE that the “NCA specifications define the protocols that govern the interaction of clients and servers, the packet format in which RPC data is transmitted over the network, and the Interface Definition Language (IDL) that is used to specify RPC interfaces. DCE RPC is based on Version 2 of NCA.”


NCA/RPCs were first described by [Dineen et al. 1987]. In their paper describing NCA, [Dineen et al. 1987] explicitly reference Cedar [Birrell and Nelson 1984] to acknowledge the origin of the concept of a remote procedure call. When we interviewed Paul Leach,
one of the original architects of NCA he confirmed that “like ANSAware, NCA includes an interface definition language (IDL), a compiler for that interface definition language that generates stubs which resolves data heterogeneity and implements presentation and marshalling in very much the same way as proposed by Birrell and Nelson in 1984”. The NCA paper also states that NCA/RPCs borrow object-oriented concepts and explicitly references [Goldberg 1983] and that it carefully distinguishes between interfaces and implementations. Moreover, the paper states that the object references that are passed as the first parameter to any remote procedure call are akin to the self concept in Smalltalk and the this concept in C++ and the paper refers to [Stroustrup 1986]. NCA/RPCs then define an interface definition language, which has many of the concepts that are still used in the IDL of OSF/RPCs and in Microsoft’s IDL. For example NCA/RPCs are the origin of UUIDs with which DCE/RPC or Microsoft IDL interfaces are uniquely identified. The use of unique identifiers in distributed computing was first proposed in [Leach et al. 1982].

Because all three RPC impact traces lead to [Birrell and Nelson 1984], we can indeed safely conclude that this is the seminal paper that is widely acknowledged as the first definitive description of remote procedure calls. It was written while Andrew Birrell and Bruce Nelson were members of the Xerox Palo Alto Research Centre (PARC). The paper describes the Cedar remote procedure call system. Cedar was a programming development environment that was developed at Xerox and RPCs were used for distributed communication between tools within this environment. It is interesting to note that Birrell and Nelson state that the particular aim of producing a program development environment forced them to define aspects of an RPC system that were only poorly understood earlier. The main contribution of Birrell and Nelson’s paper is that it for the first time determined the key architectural abstractions required for an RPC mechanism, such as stubs that marshall and unmarshall parameters, that these stubs can be derived automatically from an interface definition of some form, the different reliability semantics for remote procedure calls, and the handling of address parameters in remote procedure calls and it reported about the experimental evaluation and performance results obtained for the Cedar RPC system.

Birrell and Nelson acknowledge that ideas for RPCs had been around much earlier. Prior to his employment at PARC, Bruce Nelson completed his PhD on remote procedure calls in the Department of Computer Science at Carnegie Mellon University [Nelson 1981]. [Birrell and Nelson 1984] state that the Xerox Cedar RPC mechanism was influenced by the “work done at MIT” and reference [Liskov 1980] on principles of distributed computing. In this paper, Liskov highlights the importance of modularity of distributed systems. To achieve that, Liskov proposes a number of distributed systems extensions to CLU [Liskov et al. 1977]. As far as we can see now, there was little influence of the guardians that Liskov proposed as a CLU extension in this paper, but her case in favour of modularity and separation of interfaces and implementation of distributed system components were extremely influential and led to the inclusion of an interface definition language in all RPC mechanisms.

Andrew Birrell and Bruce Nelson describe the importance of an interface in [Birrell and Nelson 1984]. It is from such an interface definition that they were the first to derive client and server stubs that they use in their architecture. This practice is now commonplace in all RPC implementations, it is used in COM and CORBA.

For defining interfaces, the Cedar RPC mechanism uses an interface definition language called Mesa. An early description of Mesa is provided by [Geschke et al. 1977].
ICSE 79 paper, [Lauer and Satterthwaite 1979] describe the language in more detail and describe that “Mesa has many of the attributes of a module interconnection language, as defined by [DeRemer and Kron 1975; 1976]” and that it therefore builds on the principles of information hiding established by [Parnas 1972].

9. SUMMARY
While conducting the study we observed a number of commonalities about the way technology is transferred from research into successful products. We summarize this paper by distilling these commonalities in an attempt to guide future technology transfer activities.

9.1 Technology transfer takes time
The discussion of the impact that research has had in a number of key middleware market segments fully confirms the findings of [Redwine and Riddle 1985], who concluded their paper by stating “The case studies show that it takes on the order of 15 to 20 years to mature a technology to the point that it can be popularized and disseminated to the technical community at large”. The transfer of research into widespread use in middleware technologies required an equal amount of time.

Remote Procedure Calls - 16 years: The key ideas of information hiding and module interconnection languages were published between between 1972 and 1979, the relevant publications describing research into remote procedure call approaches appeared between 1980 (Bruce Nelson’s PhD thesis) and 1987 (Apollo’s Network Computing Architecture). RPCs were included in the first operating systems (Domain and SunOS) from 1987 and standardized between 1988 (IETF ONC RFCs) and 1991 (OSF/DCE).

Distributed Transactions - 20 years: Starting from Moss’ PhD thesis in 1980, a large number of non-standard transaction mechanisms were published. The notion of transactions was first incorporated into distributed objects in Dixon’s PhD thesis in 1988. Between 1994 and 1996 distributed transaction mechanisms were standardized by the Open Group (the ODTP XA and RM specifications) and the OMG Object Transaction Service and following inclusion of the Java Transaction API and the Java Transaction Service in the J2EE specification application server products brought distributed transactions to widespread use around the turn of the millennium.

Distributed Objects - 15 years: A significant amount of research into distributed objects occurred between 1985 (Eden) and 1988 (Emerald and Argus). The work was consolidated by Andrew Birrell’s and Greg Nelson’s network objects in 1990, picked up in 1996 by Waldo at Sun, standardized through JCP in the late 1990s and has started to be used broadly with the rise of application servers from 2000.

9.2 Interdisciplinarity
We note that the impact traces we have documented above very frequently cut across the different disciplines identified in the ACM computing classification. The RPC trace involves citations from the operating systems, networks, programming languages and software engineering literature. Likewise the distributed transaction trace involves citations from the operating system, database and software engineering literature.

We also observe that the largest impact is sometimes in an industry related to a CS discipline other than the one in which an idea was first published. Gray’s notion of transactions were first published in a book on “Notes on Operating Systems”. They were then nurtured...
by the database community and eventually were integrated with object-oriented concepts
and their largest use of distributed transactions to date is in application server middleware.
Likewise the first ideas of message passing were published by Steve Reiss to support inte-
gration of software development tools and subsequently they were adopted by tool vendors
and transferred to the middleware departments.

9.3 Importance of PhD research
Another interesting observation is the significant number of PhD theses that featured in the
impact traces. We can only give some examples. For the RPC trace these include Nelson’s
PhD thesis at CMU and Panzieri’s thesis at Newcastle. For the distributed transaction trace
it includes Moss’ thesis on nested transactions that provided the basic concepts used for
nested transactions in the CORBA Transaction service and Dixon’s thesis at Newcastle
on the integration of distributed objects and distributed transaction management. For dis-
tributed objects Andrew Birrell’s and Greg Nelson’s network object paper refers to Orca
for which Bal’s thesis defined the programming model at Vrije Universiteit in Amsterdam.
Likewise the distributed object model of the Emerald system that influenced Network Ob-
jects was defined by Hutchinson in his PhD thesis at the University of Washington.

We believe it is therefore fair to state that these PhD theses provided the foundations
for many concepts and their experimental validation in prototype systems that are used in
middleware today. And we go on to postulate that without the work of these PhD students
we would not have distributed object models, RPC semantics and architecture, distributed
transactions and application servers in their current form.

9.4 People movement
Providing conceptual foundations and experimental validation, however, is only one side of
creating impact. These concepts and the knowledge about how to apply them in carefully
engineered systems need to be transferred into industrial practice. We observed that a very
successful mechanism for that transfer that may currently not receive sufficient attention is
the mobility of researchers who devised these concepts from research into industry. Again
we can only give a few examples.

Following completion of his PhD in the Cambridge Computing Lab and a period of
post-doctoral research on distributed systems with Roger Needham, Andrew Herbert was
hired as Technical Director for the Alvey-funded ANSA project in 1984. The ANSA Team
produced the ANSA reference model, middleware and a series of design papers over the
next ten years; in its latter years the project was funded by Esprit; and directly by indus-
trial partners through Architecture Projects Management Ltd (APM). Following graduation
with a PhD from CMU, Bruce Nelson joined Xerox PARC where he wrote the definitive
paper on RPCs with Andrew Birrell. Birrell had similarly obtained his PhD also from the
Computing Lab in Cambridge. Andrew Birrell then went on to move to DEC System
Research where he wrote the Network Object paper that provided the basis for Java RMI
with Greg Nelson. Andrew Watson who had worked in various European ESPRIT research
projects at APM on ANSA moved to the OMG, where he played a pivotal role in the defi-
nition of the CORBA standard. Joe Sventek moved from APM where he had worked with
Andrew Herbert and Andrew Watson to Hewlett Packard where he wrote the CORBA 1.0
specification jointly with Jim Waldo Sun. Jim Waldo then moved from Hewlett Packard
to Sun where he wrote the RMI specification. Graeme Dixon moved from Newcastle to
Transarc where he wrote the OMG Object Transaction and Concurrency Control Service
specifications.

9.5 Standardization

We note that all middleware market segments we have discussed in this article have undergone significant standardization efforts. The web services technologies have been standardized by the World Wide Web Consortium, the J2EE specification was defined through the Java Community Process. CORBA was defined by the Object Management Group and the two main flavours of RPCs were defined by the Open Group and the IETF. The Open Group also standardized the interfaces for distributed transaction processing and the International Standardization Organization played important roles both in distributed transaction processing and in standardizing SGML. We would even go a step further and say that we have not seen any significant impact of research results without the results having been consumed by a standardization process.

Apart from facilitating the agreement between the main industrial stakeholders that is required for a technology to gain widespread adoption, standardization plays a number of further important roles that may be less obvious. Firstly, standardization is a necessary, but not sufficient, pre-requisite for achieving interoperability between different technologies. This is important in order to preserve investments, to achieve migration paths between technology adoptions and to facilitate large-scale integration. Secondly, standardization bodies are a technology transfer hothouse. When standards are created competing proposals are prepared, written, assessed and merged. As is evident from the significant number of minutes taken during these discussions intense debates occur on the merits and shortcomings of proposed techniques and these debates achieve engagement of the participants in the subject matter. As a result technology transfer occurs very successfully during standardization. Thirdly, standardization focuses the attention of market analysts. Once significant investment is committed into a standardization process, which often demands availability of products that implement the standardized technology, analysts gain confidence to comment on the development of that technology and have done so in the past. As a result the standards and their assessments by market analysts become focal points for future investments.

10. CONCLUSIONS

In this article, we have shown the ample influence that research in different Computer Science disciplines had on the development of middleware standards and products implementing these standards. Without researchers laying the foundations in Computer Science departments at Brown, CMU, Cambridge, Newcastle, MIT, Vrije, University of Washington without industrial researchers at Xerox PARC, IBM Research, HP Labs, DEC Research, AT&T Research, APM consolidating basic research results, contributing them to standards and informing product development departments we would not have the 8.5 billion dollar middleware market in the same form as we have it today.

Acknowledgements

We are very grateful to Mike Mahoney of Princeton University, whose experience and guidance on history of science has helped us to steer clear of the many pitfalls in conducting research outside our immediate area of expertise.

This article could not have been researched without the considerable help from a large number of people. Amongst others, we are grateful to Andrew Watson, Dino Mandrioli,
Frank Leymann, Andrew Herbert, Santosh Shrivastava, Fabio Panzieri, Andrew Birrell, Joe Conron, Ward Rosenberry, Paul Kettley, Adrian Colyer, Stefan Tai, Dick Dievendorff, Paul Leach, C. Mohan, Rob Drew, Steven Reiss, Graeme Dixon, C Mohan and Brian Randell. The authors are also indebted to the members of the Impact project and in particular to the steering committee members, Lee Osterweil, Jeff Kramer, Alexander L. Wolf and Carlo Ghezzi for the guidance and encouragement they have provided during the last five years.

The authors thank ACM SIGSOFT, the Institution of Electrical Engineers and the National Science Foundation for significant financial support that made this study possible.

REFERENCES


ACM Journal Name, Vol. V, No. N, Month 20YY.


ACM Journal Name, Vol. V, No. N, Month 20YY.


ACM Journal Name, Vol. V, No. N, Month 20YY.


ACM Journal Name, Vol. V, No. N, Month 20YY.

