Online games and e-business: Architecture for integrating business models and services into online games

C. E. Sharp
M. Rowe

Online games are the future of the interactive entertainment industry. The idea of integrating business services into online games holds a number of exciting possibilities for new business models, new markets, and new growth. We describe an architecture, Business Integration for Games, and an implementation prototype, for integrating online games with business services. We also describe a demonstration system that embeds our prototype into the popular first-person-shooter game Quake II™.

INTRODUCTION

Online games, which give the player the ability to compete against other players over a network, emerged seriously in the mid-90s. This rapidly evolved from a novelty feature to an expected function by players, and game designers adopted this approach to build multiplayer (MP) and massively multiplayer (MMP) genres of games.

A key difference between these genres is scale, and with it, the associated infrastructure to support it. The MP games confine the number of concurrent players in a single game to somewhere between 16 and 32. Usually, the game can be played either stand-alone or in multiplayer-network mode, and one of the players' machines acts as the server. The game session is relatively short-lived (minutes to hours). If the server crashes, the game is, at worst, over or, at best, severely disrupted. The MMP games, though, are a very different affair. The most popular MMP games today have subscription bases in the millions, with hundreds of thousands of players online at any one time, spread over hundreds of servers. The shared game session lasts indefinitely. This requires a much more stable environment; thus, these games have to run on dedicated servers equipped with a persistent database. Network bandwidth to support the game-related traffic is also required, and all this obviously has an associated cost.

These two different genres and their associated infrastructure requirements and costs spawned two different approaches to sustained revenue generation. The first one, mostly associated with the MP games, is based on the provision of a portal to act as a hosting and matchmaking Web site for players of these games. The portal site offers either a free membership to players and generates revenue through online advertising or a premium membership, free from advertising. The members gain access to services for locating other players and games, league tables and high scores, patches and add-ons, and use of the portal's dedicated server machines for playing games. An example of such a portal is GameSpy.com,[1] which maintains a subscription-based membership and provides an aggregation point for a variety of games that run on a variety of platforms. Some game publishers run their own portal sites with free membership, dedicated to hosting their own games and ensuring a quality experience for the community of players. The downside from the player's perspective is that these sites are limited to the products of the publisher. An example of a publisher portal site is Battle.net,[2] dedicated to games from Blizzard Entertainment.

The second business model, associated with the MMP games, is based on subscriptions that include a persistent presence in the game environment. For typically ten to fifteen U.S. dollars per month, a player has access to a game character that may be developed over time to accrue additional features for a more enjoyable game experience. The reasons players continue to subscribe include a strong community spirit, exciting game experiences, and an ever-increasing investment of time and money in the game character—if you stop paying for your subscription then your character (and all the experience and wealth gained) is lost. It is not uncommon for the subscription to be continually renewed while the account is dormant.

These games belong to the role-playing game category, and are often referred to as MMORPGs (massively multiplayer online role-playing games). Game characters are often involved in adventure and exploration jointly with other players, and aim to achieve some objective and gain rewards. The more rewards gained, the more powerful the character. Virtually all of these games have some kind of embedded trading mechanism that allows players to exchange wealth among them in the game world. For instance, in one of the most popular MMP games, EverQuest**[3] from Sony Online Entertainment Inc., players assume the roles of pseudo-medieval fantasy heroes, gaining magic and gold in a land of dragons and wizards. Players are able to buy and sell their virtual property in exchange for virtual wealth, but this virtual economy is confined to the game world and is not a means by which the game service provider makes any of its revenue. Trading virtual wealth in the game world, however, has spawned a third business model that is now emerging.

From the earliest use of MMPs (Ultima Online**[4], EverQuest, Asheron's Call**[5]), the players in the community have recognized a gap in the market. Whereas some players are unable to devote the time, or lack the skill, to develop powerful characters and gain access to the more enjoyable game experiences, they are willing to pay real money (above and beyond the subscription fee) in order to acquire this virtual property. Thus, a real economy has emerged in which artifacts of the game world, such as magical items, weapons, or even whole characters, are bought and sold for real-world money. The means by which these transactions occur are often through an external medium, such as an online auction site like eBay.[6] The game service providers have historically frowned upon this practice, claiming that it is they who own the intellectual property rights to the items being traded, not the players, and that the trade is therefore illegal. But despite various attempts to prevent it, the practice is now an acknowledged side effect of the MMP game genre, and some newer MMP releases have attempted to build this into their design from the outset by providing auction functions and the ability to exchange real-world currency for virtual in-game currency. The open market, however, is a strong force, and this has not really deterred players from continuing to use external auctions and payment services.

Another reason why the game service providers dislike this real money trade is that, if left uncontrolled, it can detract from the game-playing experience, undermine overall player satisfaction, and put the service provider's continued revenue stream at risk. If it ends up that the richest players are automatically the most successful, this imbalance can be very frustrating to the other players.

An obvious problem arises from this kind of external transaction—the financial exchange is completely decoupled from the asset exchange because the two transactions occur in totally unrelated environments, and it relies on the humans in the loop to monitor and maintain the integrity of the two. This has inevitably led to cheating by unscrupulous players (and even non-players) through bogus transactions. Clearly, this is not a new problem, but one that is present in auction and e-commerce sites on the Web, where the purchase and delivery of goods are separated over time. However, the coupling of the financial transaction and exchange of assets in this case is eminently possible and would make for a much safer and reliable experience.

Meanwhile, MP games are developing further revenue channels through the development of "episodic content" that can be purchased online and used to enhance the original game experience. Since the beginning of the MP genre, a feature of these games has been extensibility. Players have been able to create their own content to augment or tailor the game. And by the same mechanisms, the game developers create official content that is released for sale. Systems, such as Steam**[7] from Valve Corporation, provide a download client system to integrate the purchase and digital delivery of the content for incorporation into the game. However, these transactions occur outside of the game itself and not between players, preventing the players from selling their own content.

Despite the two genres of MMP and MP games exhibiting different revenue models and infrastructure requirements, they both share a common feature. Players continue to play the game and pay for subscriptions or add-on purchases if the experience continues to be enjoyable and is perceived to be worth investment in the long run. If players cannot trust the other players in the community, if real-money trade results in a playability imbalance, if the content does not get updated regularly, or if the players cannot contribute to the content of the game as a whole, they will lose interest and go elsewhere.

This problem is very similar to that faced by Web-based businesses, striving to make their Web sites "sticky" through new and interesting content and through community participation, and it should not be surprising that the technical challenges facing Web portals are relevant to the evolution of online games. However, the browser is a very different operating environment and interactive experience from that of a game. Early attempts to integrate e-commerce into games used very simplistic means, usually involving the launching of a Web browser. But this approach has serious deficiencies, because launching a Web browser involves a clear disruption of the game experience and a disconnect between the e-commerce transaction and the in-game mechanics.

In this paper we describe an architecture, Business Integration for Games (BIG), for integrating business services within online games. The work was performed within an incubator project, called Aspen, that was set up to investigate this architecture and implement a prototype. The implementation involves three main components: a thin *client connector* included in the game platform, a *process broker* based on WebSphere* to act as both agent (for the client connector) and intermediary (agent for multiple clients), and a collection of business services. The prototype is available on the alphaWorks* Web site under the name Business Integration for Games.[8]

To demonstrate the viability of integrating business services with online games, we applied the BIG technology to several games, the most notable being Quake II**. This game, from id Software,[9] is a benchmark for MP games and is highly representative of the genre. Moreover, its source code is available under the GPL (GNU General Public License) open-source license, making it suitable for our integration experiments. Since this work was completed, several MMP games have become available as open-source code, making it possible to explore the application of BIG to the MMP genre.

The rest of the paper is divided into the following sections. First, in the "Business Integration for Games" section, we review the current trends for systems integration in the enterprise, we introduce the programming environment for game development, and we describe the BIG architecture. The next three sections cover the three main components of the BIG architecture: client connector, process broker, and business services. Then, in the section "Implementing the BIG architecture," we describe our prototype and its integration with Quake II. We summarize our results in the "Conclusion" section.

BUSINESS INTEGRATION FOR GAMES

In this section, we first look at the information technology (IT) industry and the systems integration work in the enterprise environment and consider the challenge of incorporating this technology into online games. We then give an overview of the game-technology and programming environment. Finally, we introduce the BIG architecture and describe its main components.

Systems integration in the enterprise and its extension to games

Over the last few years, the IT industry as a whole has moved to the adoption of service-oriented architecture (SOA)[10] and a supporting infrastructure for SOA, based on the Enterprise Service Bus (ESB)[11] and the use of Web services in particular. Reusable function is made available to the world through well-defined interfaces and by using open standards and protocols, where integration can effectively be achieved at runtime through the use of directory services to discover and determine the integration requirements. This should dramatically reduce software-development and systems-integration requirements and cycle times. In the traditional enterprise world, there are many well-publicized exploiters of the new technology. Service providers such as eBay and Google are exposing their application programming interfaces (APIs) by means of Simple Object Access Protocol (SOAP) and Web Services Description Language (WSDL) to allow other Web sites and applications to incorporate these services into their own offerings, and enterprises such as Charles Schwab & Co., Inc., are using the ESB infrastructure and Web services for enterprise integration.

Extending this approach to games, however, by making various services available (to perform generic functions such as payment services) through a Web service interface is not enough to be of immediate use to the game industry. Game developers are not interested in a new and unfamiliar layer of complexity with which they must cope in order to build their systems. The main technical challenges in applying traditional enterprise integration techniques to online games can be broken down into three categories:

- *Integration Logic*—The various members of the value chain associated with the online game service (such as providers of payment and transaction services, or providers of digital content delivery and protection services) need to interact and interoperate with each other in some way. A truly flexible solution would not require that these providers be already integrated or even aware of each other.
- *Business Logic*—The code that embodies the business constraints, such as terms and conditions of interaction between various parties, is not relevant to the game and should be separated from the game logic.
- *Security and Trust*—The execution of the business logic may involve access to private information belonging to various parties. For example, to transfer funds from one party to another, an account number and PIN (personal identification number) may be required. Consumers are reluctant to have their private information configured within the game, as they do not necessarily trust the game code (which represents the entity that may be actually charging them for its use) to manage their financial transaction. This problem is exacerbated if the transaction takes place between players. Coupled with this issue are the problems of protection from malicious attacks (either from other players or from rogue service providers) and the security of hacked games.

Each of these categories contains a complex set of issues and, although apparently orthogonal, all three categories must be addressed with a holistic approach to ensure that meeting the requirements of one category does not reduce the efficiency of another.

Programming environment for game development

The typical enterprise application developer usually programs in Java, C#, or some scripting language. Often, the code is executed in a managed environment such as a J2EE** or .NET** container. During development the emphasis is on manageability of the code, portability (i.e. adherence to standards), and the business logic that the code is implementing. The existence of a comprehensive container environment means the developers need not (indeed, may actually be prohibited from) concern themselves with execution details, such as network and transport reliability, security, thread management, and memory management. These low-level concerns are addressed by the container itself or other middleware components. External interactions with the code travel through many layers of infrastructure code and typically include a Web browser interface. These interactions may be synchronous (where the thread blocks for the response) or asynchronous. The code executes under the strict control of the container environment. Unlike the game environment, any performance concerns are constrained to the efficiency of the business logic and the scalability of throughput provided by the container infrastructure. Predictable low-latency response time is not a metric that usually concerns the developer.

In contrast, the game developer typically programs in C and C++, and even assembly language for the more time-critical components, and the code executes in the operating-system (OS) environment of the target hardware. This environment may vary from the OS library support environment as provided by Windows** and Linux** on PCs, to some RTOS (real-time operating system) environment on a console or portable device. There is no managed container environment. The typical game is still written as a non-threaded application with a single main loop. The main loop must complete its cycle with perfect regularity in order to refresh the display in a continuous smooth flow. Any unpredictability in display updates is obvious to the eye of the player and severely detracts from the game experience. Therefore, the game itself is the main execution control engine, and all calls to other libraries and OS services must abide by the constraints of the game execution timing; otherwise, they impact the game performance as a whole. The game developers need to concern themselves with most of the low-level details of execution, such as memory management, thread control (if any), and network connectivity and operation. There is an increase in the use of networking middleware, but this is still fairly low-level and involves packet management functions and object propagation over UDP (User Datagram Protocol). For MMP games, as mentioned earlier, the game client provides the player with access to a shared "world state" that is maintained and persisted across a server infrastructure. The server component of the game is still written largely using the same programming paradigms, with performance being a key issue, leading to integration of database technology as the persistent store of the world state with vendor-specific APIs, rather than open, platform-independent APIs such as ODBC (Open Database Connectivity). The database is used both for persistence of game state and player subscription management, including authentication of clients to the game. However, architecturally, the server side looks very much like a typical enterprise Web application, consisting of edge servers acting as gateways between the clients and the servers and providing routing and load balancing to the servers running the game—effectively a simulation that tracks the state (position, orientation, velocity, inventory, etc.) of objects, updated by inputs from players and artificial-intelligence agents. Often, the virtual world is either replicated horizontally across servers (effectively a series of "parallel worlds") to spread the load statically by assigning a player subscription with a specific world. This is known as a *shard* architecture—the name reflecting how the universe is broken into shards, and players can only explore the world on the server (or server cluster) that they are connected to. Newer MMPs are exploring grid architectures to distribute a single virtual world over a number of servers by assigning a portion of the world to each server. Here, as players move from one portion of the world to another, the gateway servers handle the distribution of the player interaction with world servers accordingly and route traffic to the appropriate server for the current location of the player in the overall virtual world's geography.

The BIG architecture

In order to take advantage of the benefits of a service-oriented architecture using ESB and Web services technologies in the game environment, it is clear that some bridging technology is required. This bridging technology could be a piece of middleware that acts as an intermediate layer between the game client and the intelligent infrastructure for discovering, selecting, and using the externally provided business services. However, this intermediate layer has to adhere to the stringent operating requirements of the game environment.

Figure 1 illustrates the BIG architecture. The Game Runtime Environment includes the game component, labeled Game, and the client connector. The game component consists of either client or server code. The Game Runtime Environment can reside on a PC or server platform, a game console, a mobile phone, and so on. The game component connects to and communicates with other components through the client connector, which in turn connects to a process broker (labeled Process Broker in Figure 1).

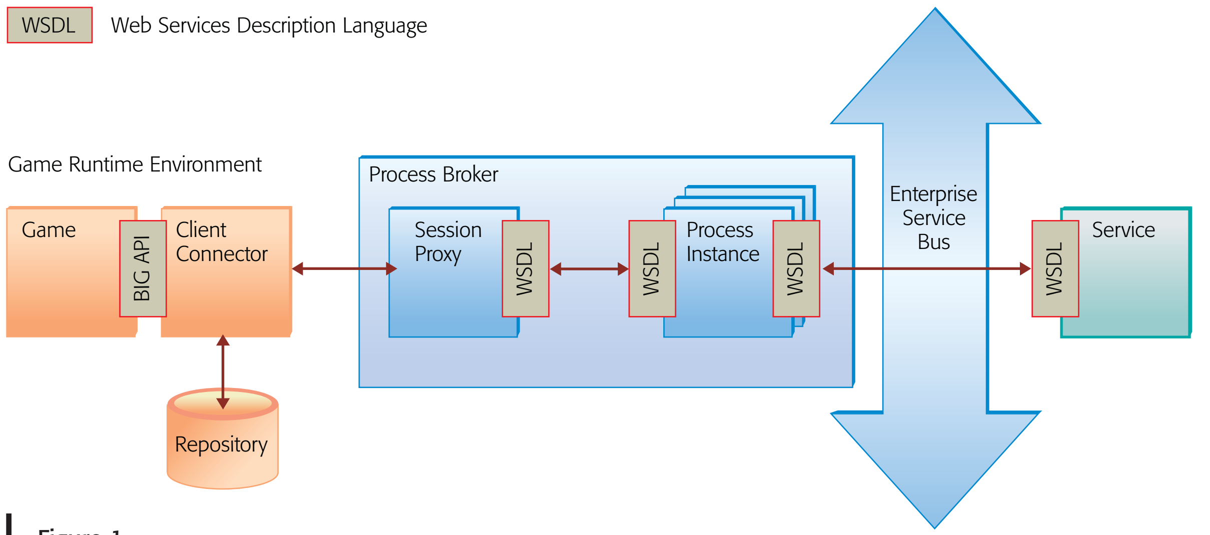

Figure 1
BIG architecture

The initial interaction between the game component and the client connector results in the instantiation of a session proxy (labeled Session Proxy) at the corresponding process broker, and the allocation of an application specific identity (ASID)—a globally unique identifier that identifies both the game instance and the process broker that owns the session proxy. The ASID is the identity that game code instances then use to refer to other game code instances when communicating with the client connector. This allows the distribution of client connectors over a number of process brokers.

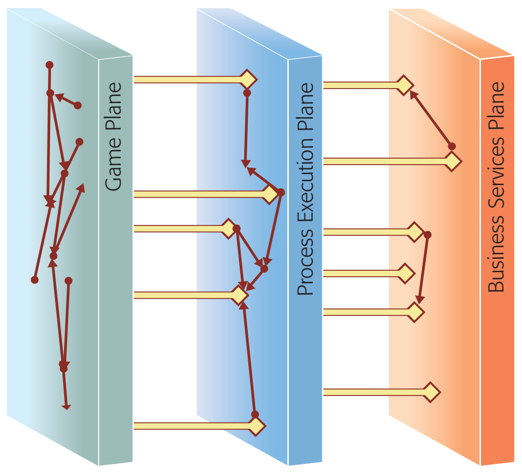

Figure 2
BIG architectural planes

The session proxy, which acts as an agent on behalf of the client connector, interacts with the process instance component through the standard Web-service interfaces (labeled WSDL). The session proxy thus acts as a Web-service facade for the client connector, to allow process instances to interact with clients by means of a standard, interoperable mechanism. This provides a level of encapsulation of the client and facilitates the use of standard process technologies, such as Business Process Execution Language (BPEL). The process instances also use the Web-service interface to communicate with business services through the ESB.

Figure 2 illustrates how the three components of the BIG architecture can be viewed as operating within three "planes," representing three separate architectural domains: the game plane, the process execution plane, and the business services plane. The game components interact within the game plane according to whatever pattern is appropriate for that specific game, be it peer-to-peer, client/server, grid, and so on. Within this plane, the various instances of the game code (players and servers) communicate according to the rules of the game (game-state propagation, game-control updates, etc.). The game components are represented in the diagram as nodes in the game plane; the edges joining them represent interactions between game components. Interactions between clients and process brokers are illustrated as arrows between the game and the process execution planes. In turn, process brokers are represented as nodes in the process execution plane, whereas edges joining these nodes represent interactions between process brokers.

Inevitably, the execution of business processes requires the invocation of business services, which operate in the business services plane.

We now examine the three architectural components in more detail.

CLIENT CONNECTOR

As Figure 1 illustrated, the client connector component is a thin client that presents the BIG API to the game code and manages personal player data by using a secure, encrypted repository. This persistent repository is available for any game that uses the client connector, so that players need not re-enter data when they acquire a new game, and their personal data is managed separately from any game. The mechanism is effectively acting as a local "e-wallet." However, due to either consumer preferences or local storage limitations, an alternative remote e-wallet service must also be considered. The client connector must be efficient, as it will reside within the game footprint and execution path. For PC-based and console games, a native, portable C version is required that can run in either a non-threaded or a threaded model.

One of the primary design objectives of our project was to hide as much complexity from the game developer as possible. The business integration infrastructure should be available for use by the game code as a kind of utility that is as simple to use as turning on a tap. To achieve this, the client connector needs to provide business functions while hiding the technical details of operation for these services. We therefore decided to design the client as a two-tiered structure, with business functions wrapping onto a microkernel. The microkernel provides the core asynchronous messaging and memory management functions for the transfer of process requests and for returning results (the responses are presented to the game code by handles to data structures). Because of the strict memory footprint and timing requirements, the client connector also provides an initialization parameter that determines the maximum amount of memory it may use. From our discussions with game developers, having predictable memory usage was as important as having predictable function return times. As the developers write their games in memory-constrained platforms without the benefit of an operating system to manage memory, they need to account for every byte used in order to fit the executable code and digital content in memory at once—there is no virtual memory and paging space to help with this. Consequently, the developers need to know how much memory each library function they use is going to consume in order to keep track of total memory usage. Having the libraries fence their memory usage helps in this objective.

The messages being passed through the client connector may become quite large, especially when digital media content is included in the message. Because it is not possible, even for smaller messages, to transfer the messages with a blocking call to the TCP/IP (Transmission Control Protocol/Internet Protocol) stack, nonblocking sockets are used. With large messages, even a nonblocking socket would exceed the maximum buffer size. The client connector kernel would therefore also be responsible for dividing the messages into chunks and delivering them over the network in iterated steps; we refer to this as "chunking."

The requirement for a single-threaded operation emerged from conversations with game developers interviewed for this purpose. The reason that game developers tend to develop games using the single-threaded style of programming is partly historical and partly practical. Games were originally developed on computing devices that typically did not have an operating system—the early 8-bit and 16-bit "micros" and game consoles. Lacking an operating system, the use of multiple threads was not a practical option. Consequently, the developers opted for the single main loop model, in which functions to be performed—to refresh the screen, handle I/O, advance state machines, and so on—are called in sequence. This disciplined approach, although limiting, leads to code whose behavior is predictable. Although newer operating systems now make multithreading efficient and reliable, there are still devices in use that do not support multithreading. To make the code more portable, therefore, single-threaded programming in C remains a popular choice for programming style.

To accommodate the single-threaded-style requirement, the client connector ensures that its continuous use of the CPU before returning control to the game code does not surpass a specified quantum, typically 16 ms. A "step" function monitors CPU use by the client connector. The clock time on entry to the function is recorded (the entry time), a small unit of work is performed ("small" being determined by our implementation), and then the current time is compared with the entry time. If the remaining execution time is not sufficient for performing another unit of work, then control is returned to the game code. The unit of work is determined through experimentation. As a result, the processing of the incoming messages (either requests to the client connector or responses to previous requests) may involve several CPU quanta. The handle-based approach to memory management allows the game component to determine whether the processing of the incoming message has been completed by inspecting the status of the handle to it. The design of the microkernel, which manages the message processing, is based on a simple finite state machine model.

The client connector is reliable and robust, so that when a call from the game component to an API function returns, the game component can depend on the middleware (i.e., the combination of the client connector and the process broker) to reliably complete the invoked function. Therefore, if the game application crashes before the result is returned, the middleware manages the transaction and either rolls back the transaction or stores the result until it reconnects with the game application.

The client connector offers to the game developer a simple API that includes meaningful business-oriented functions, such as `charge` and `trade`. The BIG API differs from the more traditional middleware APIs, which present to the developer network-oriented functions, say, for sending and receiving messages, and leave it to the developer to construct the right sort of messages and send them to the right sort of network endpoints. The business-oriented functions of the BIG API encapsulate a number of message exchanges that are required for performing the business function. These functions wrap the microkernel messaging functions and reflect the business process they are associated with. For example, a `charge` function call supports the business parameters for executing a charge transaction—the identities of the payer and the payee and the amount charged in a specified currency—and marshals these details as a request message to the process broker. The result of processing the request, the return value of the `charge` call is returned asynchronously to the game component by a data structure to which an access handle points.

Quite often, the business process involves the collaboration of additional parties, not necessarily business service providers. For example, the `charge` function involves both a payer and a payee. Clearly, it should not be possible to charge the payer's account without that party's consent. Consequently, the request initiated by one party may result in a secondary request being sent by the business process to another party's client connector. To allow that second party to respond to the request, a matching `accept_charge` function is also provided as a wrapping function in the client connector. Parties receiving a charge request from a process broker inspect the message to determine its type and values and respond with the appropriate function call, in this case `accept_charge`, passing the message handle of the original charge request as a key parameter.

By requesting processes to be executed through the process broker, the client effectively interacts with Web services asynchronously, using these Web services to provide value-add e-business function rather than core game logic. The business processes with which the client connectors interact by means of the process brokers are themselves services that expose and utilize interfaces. It is these interfaces that are mirrored at the client connector as the function calls exposed to the game developer. ***Figure 3*** illustrates this relationship.

The diagram illustrates the process of transferring funds between two game clients. Game client A initiates the process by calling the `charge` function, part of the payment API. This results in the client microkernel creating a charge request message and sending it to the process broker. The message is received by the session proxy instance responsible for that client. The session proxy initiates a charge process instance by invoking the `charge` operation on the process broker's Web service interface for the charge process.

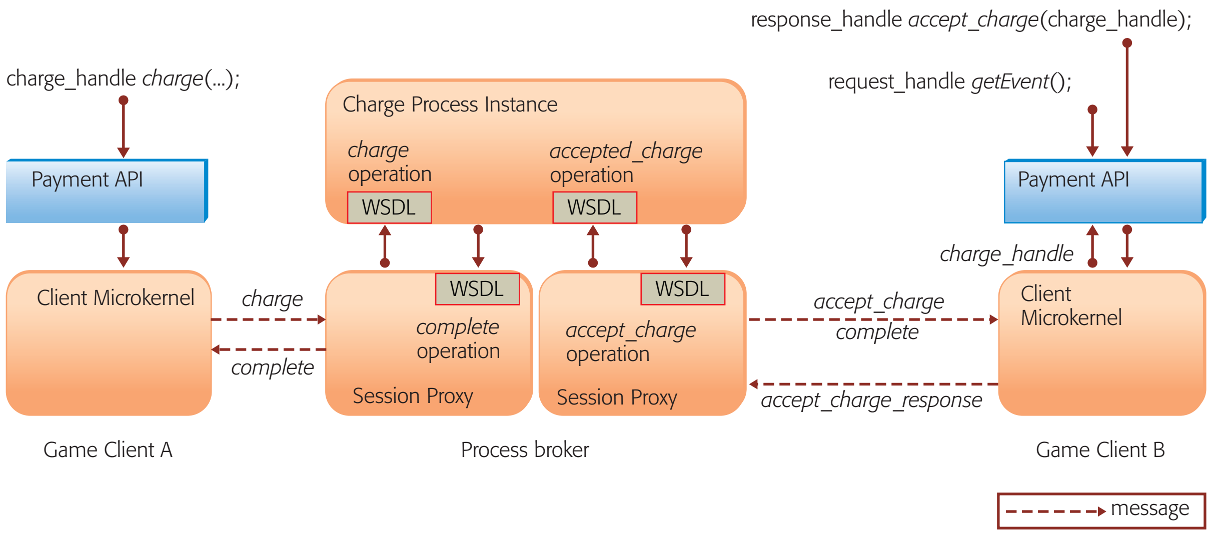

Figure 3
Client and process broker interactions

Next, the charge process seeks consent for payment from game client B. This is achieved by the charge process instance invoking the `accept_charge` operation on the Web service interface of the session proxy identified in the initiating charge request. This session proxy creates an `accept_charge` message to game client B. The microkernel at B receives this message and informs the game application via the payment API. Game client B determines whether the response is positive (it may do this by informing its user, the player, by some in-game dialog); then it replies by calling the `accept_charge` function and passing the original event handle as a reference. This microkernel creates an `accept_charge` message and sends it to the session proxy in the process broker. The session proxy for B uses the message to form the invocation of the `accept_charge` operation on the Web-service interface of the charge process.

The charge process instance then invokes the necessary third-party payment Web service (not shown in the diagram) to transfer funds from B to A, and when successful, send a `complete` message to both parties—via their respective session proxies—to indicate the process has been completed.

Security and trust

The client connector is also responsible for moving personal, sensitive data away from the game code and client API to a place where it can be administered and controlled by the owner of that data, the player. This provides enhanced privacy and anonymity for the player and engenders a level of confidence and trust in the game infrastructure.

In a Web browser-based scenario, it is common for a Web site to require authentication from a third party (such as a payment service). The acquisition of security credentials and their authentication with the third party is done through HTTP (HyperText Transfer Protocol) redirection followed by direct negotiation between the client browser and the third-party Web site, and then the transfer of the resulting authorization token to the original Web site. However, because the client connector connects only through the process broker and has no direct access to the third-party services, either from a network perspective or a protocol perspective (e.g., the third-party may use SOAP and WS-Security[12] to facilitate authentication), the client connector has to work through the process broker. It is therefore necessary to establish sufficient trust between the client connector and the process broker, and possibly to establish separately trust, by proxy, with third-party services.

Upon initialization and startup, the client connector establishes a secure session with a process broker and associates the session with the player's identity within the BIG infrastructure. This allows separate player sessions to be uniquely identified for auditing

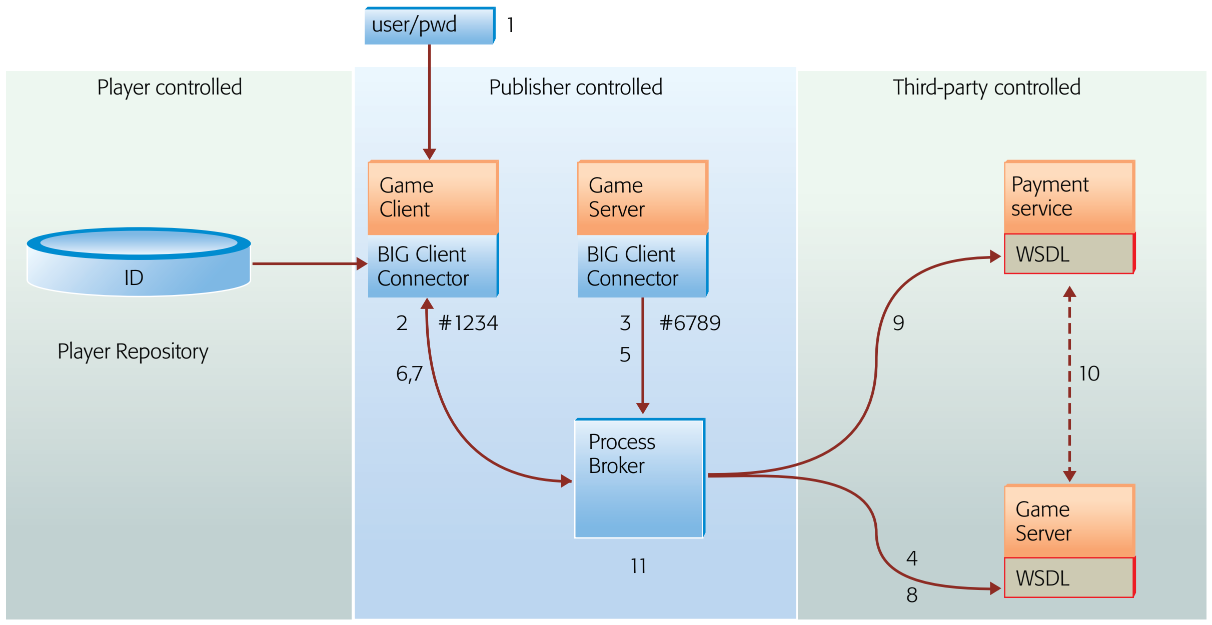

Figure 4
Security and trust domains

purposes. The client connector is performing two levels of authentication here. First, it is establishing a secure messaging channel between the game code and the process broker to ensure the confidentiality and integrity of messages passing between them. Initially bidirectional SSL authentication was considered for this, but no implementations were sufficiently granular in their operation to work in conjunction with the message-chunking approach needed. We therefore chose a simple symmetric key approach for our prototype implementation. The establishment of this session results in the allocation of an ASID that performs two functions; it acts as a token that other clients can use to refer to each other via the client connector APIs; and inside the process broker that allocated it, the ASID acts as a handle to an external address via a process broker.

The second level of authentication is between the player and the services behind the process broker. As previously mentioned, players are able to configure the client connector with their own details, such as the *userid* and *password* for their payment provider. In our simple prototype, we simply passed these details, encrypted, between the client connector and process broker, for subsequent use by that process broker for issuing a service request, such as payment authorization, to the relevant service provider. This is sometimes referred to as a trusted agent model, where players provide their credentials to the process broker to act on their behalf. However, this is not a satisfactory approach for a very loosely coupled environment, where a process broker may not be trusted to the extent required for this approach. We therefore considered a more sophisticated approach whereby any business service that is likely to be used behind a process broker would provide a security mechanism for establishing a security context with the client connector, tunneling through the process broker, and this context would then be used to authenticate individual requests to the service. Although our intention was to use the Web Services security specifications (WS-Security, WS-SecureConversation, WS-Trust, and WS-Federation) for this, the technology had not matured in time for our prototype work.

Figure 4 illustrates the security and trust domains in our implementation. The steps in the process of end-to-end authentication are as follows:

1. The player is authenticated to the client connector (passing the *userid* and *password* to enable access to the player repository). The encrypted player repository is now accessible to the client connector (but no API function directly exposes it to the game code for access).

2. The client connector sets up a secure session with the process broker (based on encrypted game credentials, such as use of the symmetric key algorithm), and ASID #1234 is allocated.
3. The same activity occurs for game server code. ASID #6789 is allocated.
4. The client connector requests the public key of the identity provider service (IdP). The client connector sends a request to the process broker to request a session token for principal “Chris” (identified in the player repository) to IdP. IdP returns a challenge nonce encrypted with a symmetric key for Chris, and the process broker returns to the client connector (a nonce is an arbitrary number generated for security purposes and used only once in a security session). The client connector decrypts the nonce, signs it, and sends it to the process broker for response to IdP. IdP sends a session token back, encrypted with the symmetric key for Chris.
5. The game server requests payment from player #1234 (Chris). The client connector at the game server issues the request to the process broker.
6. The charge process executed by the process broker requires Payment Service authorization from client connector #1234 and sends a *charge* event to the client connector. The client connector for Chris indicates an incoming event to the game code. The game code inspects the message and sees it is a charge request from #6789.
7. Chris’s game client makes an `accept_charge` call on the client connector. The client connector issues signed proof over a time stamp plus data to the process broker by using a session token from step 4.
8. The process broker uses this to request `SecurityToken` from the identity service for Payment Service. The request is issued based on the signature from step 7 on behalf of principal Chris. IdP responds with a session token for Payment Service.
9. The process broker uses this to authorize payment. It requests an authorized charging session token from Payment Service, using the token from step 8 as authentication.
10. Payment Service *may* check authorization of the principal with the IdP attribute service.
11. The payment security token is discarded (stale).

It is hoped that in the future this mechanism will be fairly easily implemented by using a combination of Web Services protocols and the client-connector-to-process-broker security mechanisms.

PROCESS BROKER

The client connector needs to establish a session with a process broker before any business processes can be invoked. The process broker is intended to be as stateless as possible, effectively acting as an edge server for the execution of generic business processes; a large, distributed network of process brokers is envisaged to provide a localized and load-balanced point of entry for a client connector into the process execution plane.

Figure 5 shows how a network of process brokers may support the interaction between game components and service providers. Each game component, which connects to a unique process broker, is allocated a globally unique ID (the aforementioned ASID) so that the clients may refer to each other in function calls to the BIG API. The dotted arrows between process brokers in Figure 5 represent logical interactions between these components as process instances are created. These logical interactions, however, are carried out via communication lines depicted as solid lines. Effectively, from the perspective of any one process broker, other process brokers (and the process instances and session proxies within them) appear as services supported by the ESB.

Considering the charge example, a game client in Figure 5 may initiate a request to transfer funds (a payment for a game asset, for instance) to another game client, supplying the two ASIDs as parameters. The identity of the Payment Service provider is retrieved from the appropriate client connector’s repository, and that client’s process broker initiates the request for service.

The process brokers act as neutral intermediaries that coordinate message exchanges between client connectors, whether they are player-to-player or player-to-game-server interactions. In some respects, they act as a generalized escrow service brokering the information associated with the transactions, and they are responsible for the reliable execution of the process and persistence of any data over long-running transactions.

The process brokers also contain the business and integration logic required to interact with the service

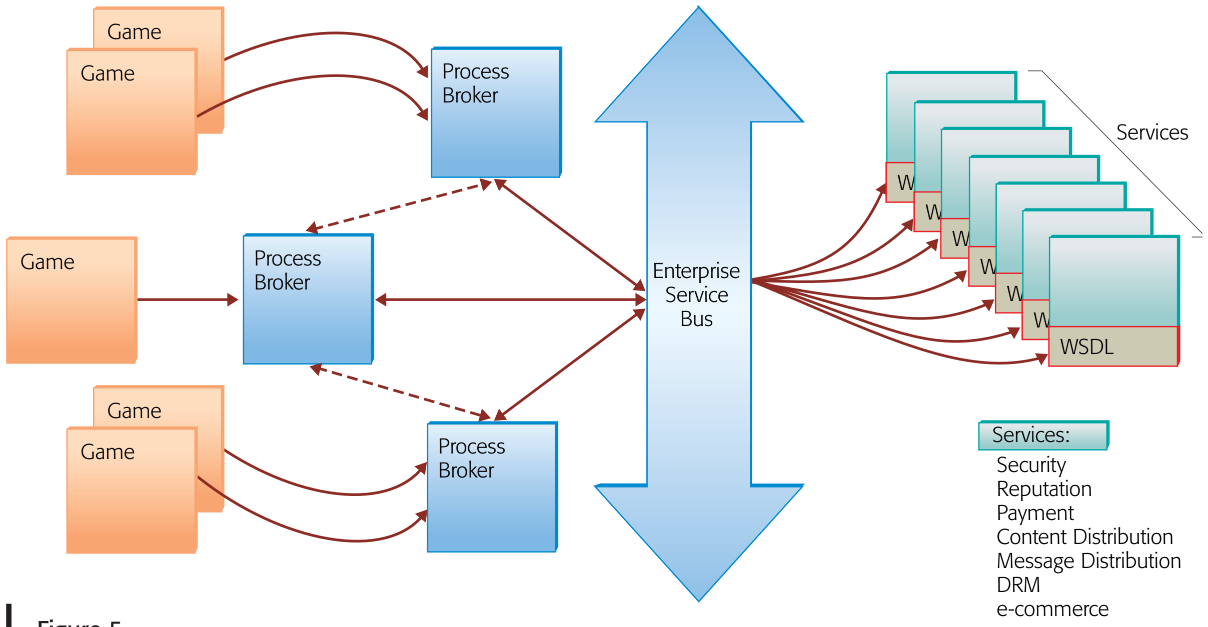

Figure 5
Network topology in a BIG implementation

providers, such as a full Web services stack, insulating the game developer and code from service details and the need to perform business logic at the client. This partitioning of game logic from business logic, both in terms of programming model and execution environment, is quite different from the current game architectures, where business models and policies are coded directly into the game code.

Because games are eminently susceptible to hacking, any business logic embedded in the game client is likely to be compromised. It is common practice, therefore, to locate business logic in the game server. However, this is only practical for games that have a permanent, trusted server infrastructure, such as MMP games have. For the peer-to-peer MP architecture games, this makes the embedding of any business-critical code in the client very problematic. The model that BIG provides, where the execution of business logic is separated into a third-party service (the process broker) that is orthogonal to the infrastructure of a particular game, is an innovative approach to solving this problem. The only code in the client, therefore, is the code that handles the initiating of requests and responses between other parties via the process broker, and only the data sensitive to that client flows in and out of that individual's client connector. Any hacking would not reveal any secrets unknown to the hacker, and tampering with the message handling would only interfere with the message flows to the process broker—which would likely result in the relevant business process not being completed. Because there is also an interprocess communication going on between the two interacting parties within the game's own architecture, any successful hacks to obviate the execution of a business process would need to be carried out at both parties' clients. Effectively, the BIG process brokers are acting as the mutually trusted intermediary infrastructure that is missing in the peer-to-peer game architecture.

The business logic is encapsulated within the process brokers as modular components that can be composed into hierarchical business processes. Although we just coded these business processes in Java** for the prototype, the intention was to use BPEL (Business Process Execution Language) to model these processes; however, the WebSphere Process Choreographer technology (IBM's BPEL engine) became available too late for our project to recode the prototype.

BUSINESS SERVICES

Business services are required to support the execution of business processes within process brokers. These services are expected to be provided as reusable business functions and implemented as

Web services. The BIG architecture provides a framework for connecting the various participating members of a value chain required for a given game scenario. These services can be thought of as plug-ins to this framework, and it is therefore expected that well-defined interfaces for well-defined business services will be needed to populate the framework.

Wherever possible, existing standards specifying service provider function should be used. For instance, the PayCircle[13] initiative is one standard that defines how a payment service provider should present its interfaces with Web services and Java interfaces. However, when these standards either do not exist or have not been widely adopted, alternatives must be sought. These can be either generic interface definitions that attempt to make likely function for a given service type canonical (e.g., payment service, asset service, security service, etc.) or specific integration logic for a specific service provider. The use of the ESB architecture can facilitate all of these different integration approaches in a single coherent administrative manner.

We now describe the service types that we defined in the initial architecture. These are basic types that can be used in combination to support complex business processes, such as trading of digital assets protected by a digital rights management (DRM) system in return for payment.

Payment service

The payment service is used whenever a payment is made by one party to another. When an exchange of funds takes place between two players (we refer to it as the peer-to-peer charging model), there is manual intervention at both ends. In the traditional consumer-merchant exchange of funds, the merchant end may be automated. Various commercial institutions are competing to provide payment services: telephone companies and mobile network operators, prepay card-system providers, Internet service providers, utility companies, credit/debit card services, and so on.

We expect that a single payment service would be the player's preferred means of making payments for items such as subscription fees and content or premium services, so that all charges can be consolidated to a single payment channel for the player's convenience. The player could choose a preferred service provider, establish an account with this provider through some external means, and then configure the client connector repository with the account and security details.

Commerce service

The commerce service provides the functionality of a Web-based store and catalog service, which typically allows customers to browse catalogs, choose items for purchase, and use shopping-cart facilities to make purchases. This functionality is stripped of nonessential details and integrated into the game environment, resulting in simple function calls that drive shopping-cart and catalog-browsing processes.

Because the BIG API implements an abstracted notion of a store, both virtual and real purchases are possible from within the game. For instance, an online store that allows the download and purchase of digital game content can be integrated into the game environment so that in a first-person shooting game the store appears in 3D graphics as a game entity with shelves stacked with weapons, ammunition, and armor. An item picked from the shelf by the player is placed in the "shopping cart." Upon leaving the store, the player pays for the contents, which are subsequently added to the player's assets.

An online store that sells physical goods, such as branded merchandise, pizza, and so on, also can be embedded in the game in exactly the same way. The delivery of purchases, however, is by shipment to the player's home.

Content service

The content service is responsible for storing and managing the digital content within the network. For example, games may inject content and make it available to other game clients and servers.

Figure 6 shows two game clients and a game server interacting via a process broker. New content injected into the game environment is made available to other clients and servers through the process broker, which uses a content management system exposed as a service. The details of how the actual content is stored and retrieved are not visible to the users of the service, so that a content repository of choice can be used to actually store the digital content. Because the game server controls

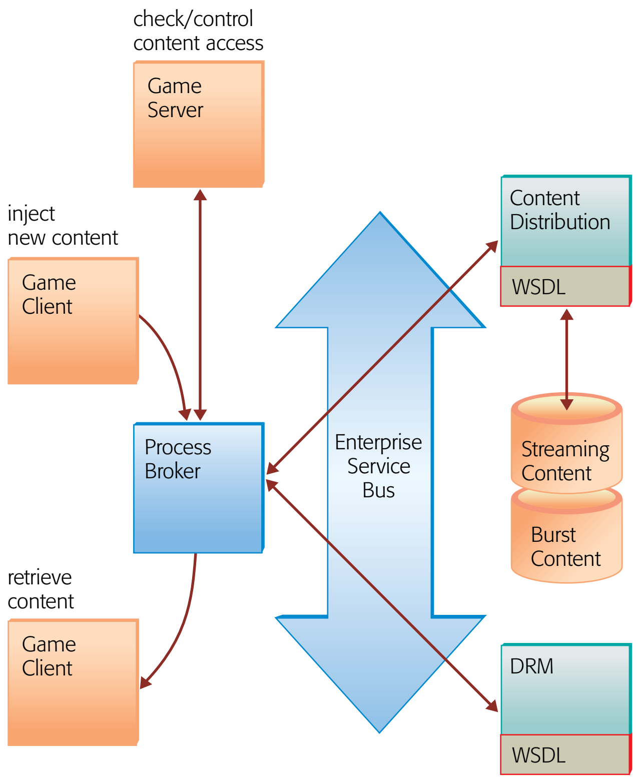

Figure 6
Content services

access to the media, the game developer is able to offload the content-management and distribution problem to the middleware infrastructure. Because digital assets are handled like any other artifact by the middleware, they can be manipulated in conjunction with other artifacts in the same transaction. For example, the game client can retrieve content in return for payment, handled in a single function call. The function call results in the process broker executing a process that groups an asset manipulation by the content service with a financial transaction by the payment service.

A separate DRM service can be used to deal with ownership issues, key exchanges, and so on, and a separate DRM clearing house can be implemented. Due to resource constraints, we did not get a chance to investigate this very interesting set of services in our prototype effort but believe this area is a very rich one for future research.

Message service

The message service is an abstraction of a message distribution hub, or message broker, that enables applications and data to be loosely coupled via a publish/subscribe message distribution pattern. This service is a built-in feature of an ESB and a natural extension of the BIG architecture.

Within the publish/subscribe messaging pattern, messages are associated with *topics*. This is essentially a hierarchical namespace with some semantics implicitly associated with it (known as a *topic space*). A message broker provides a logical "cloud" that represents the topic space of all topics. Entities using this cloud can either produce or consume messages, and they do this by *publishing* a message to a topic, or *subscribing* to a topic. The use of wildcards is permitted when referring to topic names. Entities are unaware of each other's existence, and they are only concerned with the topic space to which they are publishing or subscribing. Each entity may actually interact with the topic space over a different mechanism, but these details are hidden from the consumers of messages.

For example, using a live tennis tournament scenario, one entity may publish messages (sometimes called events) to a topic */Sports/Tennis/Events/Wimbledon/Matches/Henman* that contains the current scores for tennis player Tim Henman in an ongoing series of matches at the Wimbledon tennis tournament. Another entity may subscribe to the topic */Sports/Tennis/Events/Wimbledon/** and get all messages published about any of the players in any of the matches in the Wimbledon tournament. ***Figure 7*** illustrates how an event distribution service connects producers and consumers of events, such as live sporting events or Web applications, within game environments.

At the far right of the diagram we see a variety of devices producing and consuming event-related messages through the event distribution service. Remote sensor and telemetry devices report information such as location, orientation, and velocity of objects, such as a racing car or tennis ball, in a live sporting event. Mobile phones receive important status messages about events within an ongoing MMP (e.g., "Your castle is being raided by the goblins") in which the user is not currently participating. Data from an ongoing MP tournament (e.g., current number of players and game statistics) are fed to a Web application and then rendered to the user by a browser.

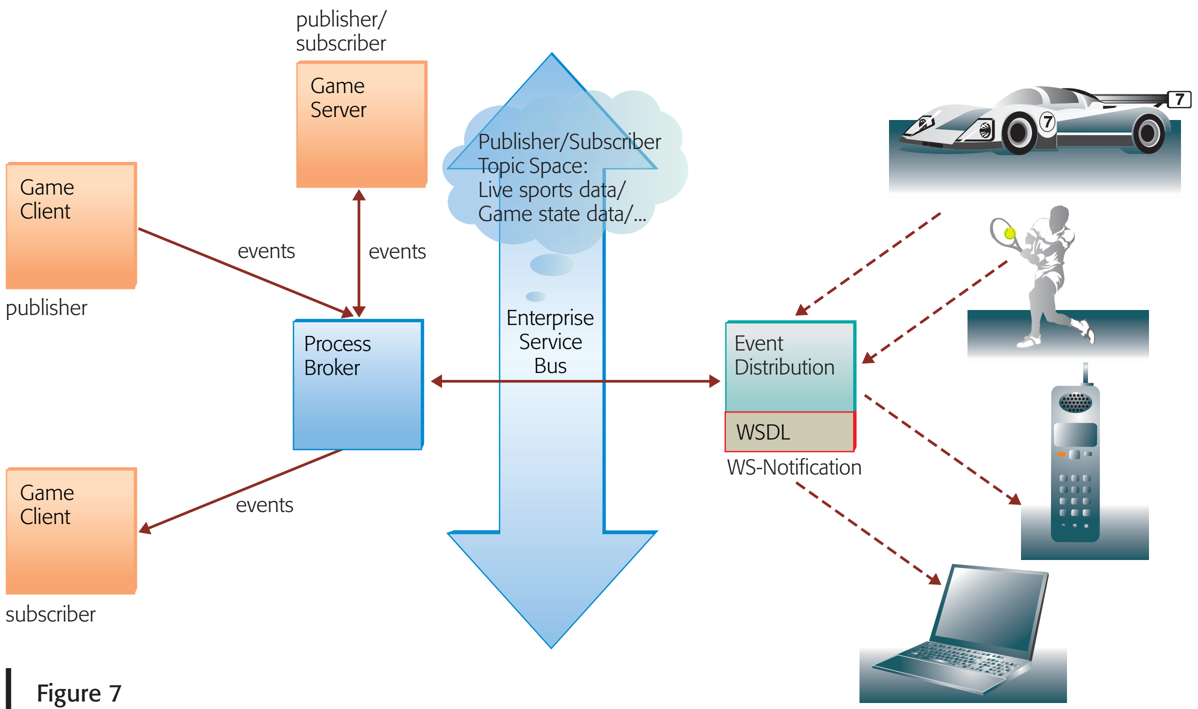

Figure 7
Event distribution service

The game code can either produce events by publishing to the publish/subscribe "cloud" or listen for events by subscribing to specific topics. This way a game could take the feed of telemetry data from racing cars via trackside sensors in a live motor-racing event and use them to simulate the actual event, without needing to understand how to interface with that particular telemetry feed.

Using the paradigm in the opposite direction to push virtual environment events out to the real world, games with multiplayer attributes such as tournament games could publish the current in-game statistics, such as current scores, player health, location, and so on, and other applications such as Web sites could subscribe and make use of this data.

This kind of integration is sometimes known as an event-driven architecture, and it is a complementary approach to SOAs. As an adjunct to our prototype work, we sponsored a project (called Event Horizon) involving college students who looked at enabling the virtualization of a topic space over a number of message brokers and the appropriate exposure of a messaging service at the client connector. The students involved in the project developed a very innovative real time strategy (RTS) game that uses the topic space as a mechanism of sharing game state, not only between instances of the same game, but also between instances of different games. It thus provides a higher-level strategy game that involves the players coordinating and deploying troops and resources over a battle-field map. Individual skirmishes between opposing troops are then realized by instances of a first-person-shooter game, such as Quake II. This was achieved by instrumenting Quake II to expose its configuration and state via topics. The high-level RTS game inspects the progress of Quake II instances and configures new ones based on actions in the RTS, via the appropriate topics.

This approach to game development and the inter-locking of actual instances of different games is certainly a very radical departure from current practice, and we feel this also presents a very rich vein of research.

Clearly many other kinds of service could be defined and integrated into the middleware, and then combined with each other to provide value-add processes usable from within the game environment. The set of services described in this section is a first pass at what might be of immediate value, but imagination within the game industry can help to define further service types.

IMPLEMENTING THE BIG ARCHITECTURE

In this section we discuss our implementation of the BIG architecture. In order to verify our implementation of the architecture and to demonstrate that it can be successfully integrated into computer games, we also describe a demo that integrates the popular first-person-shooter game Quake II into BIG. Although, given its "fast and furious" nature, Quake II was not necessarily the best vehicle for demonstrating the BIG concepts (some sort of MMP game might have been a more suitable choice), Quake II had the advantage of being immediately recognizable, and, more important, its source code was available under the GPL license. This made it possible not only for BIG technology to be integrated into the game, but also for the game to be modified to properly exploit that technology, and, in doing so, to demonstrate a number of potential business models.

Of most immediate importance to the game developer, and from an integration point of view, are the BIG APIs. As discussed previously, these are a set of process-oriented APIs, each of which provides access to one of the business services described earlier, and each of which uses a small set of simple verbs to provide access to the complex functionality offered by those services. The focus of the incubator-funded project was on the payment and commerce aspects of the business processes, and so the APIs that were developed concentrated on these areas. The resulting APIs use very simple but powerful process-oriented verbs (such as, simply *charge* on the Payment Service API) that allow the game developer to access the back-end services with minimal effort and without having to worry about the underlying business and integration logic for those services.

Given the implementation requirements just described, and because C was the programming language of choice (or at least the lowest common denominator) for most game developers at the time, the APIs were written in C. Due to the previously described constraints of running in a high-performance gaming environment, all but the most basic API functions were designed to behave in an asynchronous manner, so as to not interfere with the flow of execution within the game.

Providing the actual implementation of the APIs is the client connector. Given both the performance and language-support requirements, this was implemented in C as a thin client with an unthreaded, finite state machine-based design. In order to not interfere with game performance, the client connector also uses its own memory heap, which is configurable by the game developer. To further reduce performance overhead, the client connector also contains no real business logic; instead, its main role is to offload the work generated by calls to the API onto the process broker.

The process broker itself was implemented to run within WebSphere and included support for the payment and commerce service functionality described by the client-side APIs. Although integration with real-world payment and commerce services had been achieved, for stand-alone demonstration purposes simple payment and commerce services were also implemented as EJBs** and run within WebSphere.

Integration with Quake II

Given that the Quake II source code was available, integrating Quake II was a relatively simple matter, which was performed by just adding the necessary API calls at the appropriate locations in the code. However, given the "jump in and start shooting" style of game play, modifications to the game engine were required in order to restructure the game play, and to allow the necessary business logic to be inserted. This was, perhaps, slightly inelegant, but ideally, any real-world game developer would not be attempting to retrofit business logic to their (potentially unsuitable) games; instead, they would design their game with the business models in mind from the start. Along with the necessary code modifications, a custom "map" (i.e., the metadata describing a game level) was also constructed, which allowed the desired business models to be demonstrated in a visually clear manner.

The Quake II demo contains a number of business models, all of which were implemented mainly for illustrative purposes. It is not suggested that all (or, indeed, any) online games should actually implement the same configuration of business models described here. Nor are these the only business models that can be implemented. Rather, these are examples of what can be achieved using the BIG approach, demonstrated within the familiar setting of Quake II.

We modified Quake II by having the player enter the game world without any weapons or usable items, and thus without the ability to actually enter combat—players arrive in an area (a central tower) isolated from the main combat arena. This is akin to the manner in which most MMPs begin, in which a player's character begins with no assets and must be built up over the course of the game. Starting the game in an isolated tower is also similar to the concept of first entering a "lobby" and chatting with other players (to discuss strategy, say) before joining the game proper. This is also the starting point for demonstrating the business models incorporated into the demo.

Content purchases

Because Quake II is a combat-based game, entering the game with no weapons or equipment puts the player at a severe disadvantage. To rectify this, contained within the tower is a "store," from which players are able to purchase weapons and equipment before entering the tournament game.

Upon entering the store, players are given the opportunity to purchase weapons, ammunition for those weapons, and other miscellaneous items (such as armor). The contents of the store are populated by making use of the BIG commerce API to query a commerce service, which maintains a "catalog" of the items that are available for purchase. As the player wanders around the store in the 3D game environment and selects items, these are added to the player's "shopping basket," ready to be purchased. In much the same way as online stores such as Amazon.com, Inc. work today, the items are not actually purchased until the player confirms that he or she wishes to purchase the contents of the shopping basket.

Of course, the ability (or necessity) to purchase weapons and equipment within the gaming world would perhaps be off-putting in a real game—games could soon degenerate into "survival of the richest." Nevertheless, there is plenty of in-game content that it would make sense to charge for; for example, access to new "levels" within a game, rather than having to purchase the latest expansion pack from a physical, real-world store. Additionally, there is a fast-growing trend in consumers wishing to customize their experiences, as seen in the enormous mobile phone ring-tone business, where content is readily purchased for small sums of money in order to tailor the appearance of the player in the game.

More important, this shows that the purchase of content can be made within the gaming world, without having to leave the game (to use a Web browser to make the purchase), and therefore without having to destroy the immersive experience that the gaming world provides.

Furthermore, the purchase of content need not be limited to digital items. Because the BIG commerce API is merely retrieving a list of items for purchase from a commerce service, it is perfectly feasible for that commerce service to be selling real-world items. For example, as demonstrated by Sony's recent updates to EverQuest 2, the commerce service could be providing access to a store like Pizza Hut, Inc., thus allowing players to purchase a pizza and have it delivered to their home without having to disengage from the game at all.

Thus, not only is it possible for online commerce to be seamlessly integrated into the gaming world, it also becomes possible for any item—be it physical or digital—to be purchased from within that gaming world.

Pay to play

Having selected weapons and other necessary items, the player is about to enter the gaming arena itself. In order to gain access to the arena and be able to play the game itself, the player must pay an access fee to the game service provider. From the store the player steps through a door that leads to the game arena and at this time is shown the charges that must be agreed to before he or she can join the game. Both the game entry fee and the store purchases are paid for, and the appropriate amounts for each are debited from the player's business account by means of the business services provided by the BIG APIs. This is just a feature of the implementation of the demo, and there is no reason why the purchase for the items from the store may not be made before players make their decision to actually enter the game. Furthermore, despite appearing on the same display screen, the two different purchases are indeed handled differently.

Once the user decides to accept the total charge, two different charges are made (and two separate charge processes are initiated). The first charge is made via the commerce API, which invokes a commerce service. For the second charge, the one-time game entry fee, the payment API is used instead.

This separation of the various charge types demonstrates a degree of revenue-splitting capability. In this demonstration, although the user only makes one charge, the payment is actually sent to two different parties: the first part of the payment is received by the commerce service provider in exchange for items purchased in the store; the second part of the payment is received by the game service provider. These two services therefore need not be provided by the same entity.

Furthermore, it is quite possible—and, indeed, preferable—for more complicated revenue splitting to occur in the back-end services. Although not demonstrated here, it is possible for the simple, one-time charge via the payment API to actually be split by some business process hosted by the process broker, with various percentages of the profit being sent to various parties (e.g., developers, publishers, etc.). This is where the real value of the BIG APIs would become evident—the game developer need add one simple verb (e.g., *charge*) into the game code, without needing to worry at all about the business logic controlling which entity receives which percentage of the actual payment. What's more, that business logic could actually change (by altering the functionality and behavior of the services offered by the process broker) without actually having to change game code.

Tournament play (winner takes all)

One final business model illustrated by the Quake II demo involves rewarding the winning player in some way. In this case, the reward is financial, with the winner receiving a percentage of the total tournament revenue received by the game-hosting service (and obtained from charges made upon players wishing to join the game). In the demo, the winning player receives 90 percent of the takings, and the game-hosting server keeps the remaining 10 percent as profit. This provides players with an incentive for joining a particular game, simultaneously allowing the game host to make a profit.

Once again, the actual business logic that controls the percentage of the initial takings that players receive as their winnings may be completely contained within the process broker. Furthermore, the fact that two different APIs are used to charge the player—and two completely independent services utilized—means that the winner's prize need only be contributed to by those parties that actually wish to. For example, whereas the host of the game service wishes to provide players with an incentive to play on his or her server in order to obtain a source of income, the commerce service provider has no such concerns because that income is obtained from the sale of necessary in-game items (such as weapons). The use of two different APIs means that the two charges are completely separate and that the commerce service provider may therefore keep all of the profit from the sale of in-game items. Thus, there are many ways of splitting revenue streams and separating the various financial concerns of all parties involved.

Overall, the Quake II demo demonstrates that, not only is it possible to bridge the programming and operating gap between current game development methodologies and enterprise computing, it is also possible to make the new environment easy to use. The demo also indicates that third-party service providers might find it easier to expose their services to other markets by integration with an open, standards-based integration framework.

CONCLUSION

In this paper we describe an architecture, Business Integration for Games, and a corresponding implementation for integrating online games with business services. We also describe a demo that integrates the popular first-person-shooter game Quake II into the BIG implementation.

Throughout the course of the project, we made some interesting discoveries about the game industry, such as the variety of business models they employ today and the evolutionary period the industry is going through that will necessitate new and more flexible business models in the future. As an industry, it is an intriguing hybrid, an offshoot of the IT and entertainment industries, and it has close affinities with both, but it is also unique. It is also a relatively young industry, and its rapid expansion over the last decade is accompanied by a shift from the custom software industry to online services.

Since we performed the work described here, Microsoft[14] and Sony Online Entertainment[15] announced game-related support for e-commerce systems that are consistent with our approach. It is not clear that they have gone as far, however, to build an open, standards-based framework to support a dynamic business environment.

The seamless integration of real world commerce with the fantasy environment of a game, which players are attracted to by the very fact that it is not the real world, involves a delicate balance to be kept. An enjoyable game experience is the critical success factor for any online game, and if the insertion of business aspects into the game scenario detracts from this experience by jarring the "suspension of disbelief" in the narrative, then it is counterproductive. This is similar to the challenge in the film industry in the way corporate sponsorship of products needs to be very carefully and sensitively dealt with in the context of any given film. The challenge of attaining a smooth integration of business function and online games is one that faces game designers and one that we do not address in this work.

ACKNOWLEDGMENTS

The authors would like to thank Rob Smith, Andrew Reynolds, Brian Innes, Arthur Barr and Steve Wallin for their tireless efforts in the Aspen Project, and James Russell for his vision and support of the project as a Software Group Incubator. The authors would also like to thank the Hursley Laboratory for supporting the project in its initial exploratory stages.

CITED REFERENCES AND NOTE

1. *GameSpy*, IGN Entertainment Inc., http://www.gamespy.com.
2. *Battle.net*, Blizzard Entertainment, Inc., http://www.battle.net.
3. *EverQuest*, Sony Online Entertainment Inc., http://eqlive.station.sony.com.
4. *Ultima Online*, Electronic Arts Inc., http://www.uo.com/.
5. *Asheron's Call*, Turbine Inc., http://ac.turbinegames.com/index.php.
6. *eBay*, eBay Inc., http://www.ebay.com.
7. *Steam Content Distribution*, Valve Corporation, http://www.steampowered.com/.
8. Business Integration for Games, IBM Corporation (2003), http://www.alphaworks.ibm.com/tech/big/.
9. *Quake II*, id Software Inc., http://www.idsoftware.com.
10. See, for example, SOA and Web Services, IBM Corporation, http://www.ibm.com/developerworks/webservices/ and *IBM Systems Journal* **44**, No. 4 (2005).
11. M.-T. Schmidt, B. Hutchison, P. Lambros, and R. Phippen, "The Enterprise Service Bus: Making Service-Oriented Architecture Real," *IBM Systems Journal* **44**, No. 4, 781–798 (2005).
12. *Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)*, A. Nadalin, C. Kaler, P Hallam-Baker, and R. Monzillo, Editors, OASIS Standard 200401 (March 2004), http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf.
13. PayCircle Consortium, http://www.paycircle.org/.
14. Xbox 360, Microsoft Corp., http://www.xbox.com/en-US/xbox360/.
15. Station Exchange, Sony Online Entertainment Inc., http://stationexchange.station.sony.com/.

Accepted for publication September 1, 2005.
Published online January 18, 2006.

C. E. Sharp (Chris)
IBM Software Group, Hursley Park, Winchester, Hampshire, SO21 2JN, UK (sharpc@uk.ibm.com). Mr. Sharp is a Senior Technical Staff Member in the WebSphere organization within Software Group; he is a Master Inventor and a member of the IBM Academy of Technology. He works as a software architect on WebSphere product support for Web services and has extensive experience in business integration technologies and issues. For his work in the field of Web services, he received an IBM Outstanding Technical Achievement award in 2004. He led the team that developed the Business Integration for Games middleware, a prototype technology that integrates online games and e-business and that is available on the alphaWorks® Web site. He is a Fellow of the British Computer Society.

Martin Rowe
IBM Software Group, Hursley Park, Winchester, Hampshire, SO21 2JN, UK (mrowe@uk.ibm.com). Mr. Rowe is a software engineer at IBM's Hursley Laboratory and is currently working on the SCORE project, a document management system that supports compliance with governmental regulations in the life sciences. Prior to this he worked on the IBM Business Integration for Games project, in which he was responsible for developing the client-side APIs and the client connector and supporting Quake II™ integration efforts. ■