Microsoft's vision for XML

Adam Bosworth Allen L. Brown, Jr.
Microsoft Corporation
http://msdn.microsoft.com/xml/

Abstract

The primary premise of this paper is that, with the advent of the web, it is now possible to design a truly simple, flexible and open architecture which allows all applications, on all machines, on all platforms, to interoperate. Another key premise is that this architectural latitude will only be possible through the aegis of logical views where, conceptually, cooperating applications interoperate through agreed upon logical conventions. This contrasts with their acting directly upon either the database tables or methods of one another. We argue that XML is the central building block for such an architecture. Moreover, several key XML schemas and conventions will be required for this revolution to be realized. We devote the remainder of this paper to spelling out the various building blocks Microsoft believes are required.

1 The Vision

Applications on the web will be easy to make open and interoperable. Goods and services will be easy to find. Any customer will be able to (for example):

- discover all sites that have some used book s/he is seeking. Then order the book from one of the sites.
- vacation in another country for a month, and easily discover who can cut her/his lawn back at home on Saturdays. Then engage someone to do so.
- open a spreadsheet or her/his own custom pages or Java applications, and easily let any of them talk directly to any site that manages her/his portfolios. Then make changes to those portfolios.

In short, it will be easy to discover and interact with data and applications on the web.

2 Why should we believe this?

Whenever one thinks about a new architecture, one should consider what has already worked (or not). If the architecture is as sweeping a one as that required to allow applications to interact across the web, then it is reasonable to take lessons from the successes of the web itself, and as well as from the successes of server-based applications, both two- and three-tier.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

3 So, what *have* we learned from the web?

First and foremost, we have learned that it isn't enough for something to be possible. It must be easy, open and flexible. The web predates HTML of course, but until the advent of HTTP and HTML, it didn't really explode. Why? The answer, succinctly, is empowerment. Once HTML and HTTP arrived, more people could create and use applications more easily. The implementations were not necessarily optimal from the point of view of performance or even robustness. They were optimal from the point of view of ease of getting started. In short, they were "drop-dead" simple.

Many people point out the deficiencies of HTML, noting especially the sloppiness of its grammar. This is true, and even regrettable, but the fact is that the simplicity of the HTML model, where tags were not used abstractly, despite what every SGML book preached. That they were used concretely to describe intended look and feel made HTML immediately approachable. The fact that HTML simply ignored unknown tags made it easy, as well, since mistakes were silently ignored.

Now, as we all know, the lack of formality led to a mess in which few can implement an HTML engine since it is expected to maintain perfect visual and scripting fidelity with what is, essentially, an unwritten, arbitrary, and complex standard. We should also learn from this and retain the simplicity without the mess. Nevertheless, the key point is that HTML exploded because it crossed some threshold of simplicity, making easy things easy, and hard things possible. The lesson is that *simplicity* and *flexibility* beat *optimization* and *power* in a world where connectivity is key.

There is a second lesson which is key. Applications need to be assembled from (coarse-grained) components that can be dynamically loaded rather than from single, large monolithic blocks. In the HTML world, these components are pages. But this lesson applies more generally to applications no matter what the basis for componentization. The reason for this is simple. The application starts more quickly, only consumes the resources it really needs, and, most importantly, can be dynamically loaded from the web. Why is this so important? It is important because of the need to *deploy*. Applications that can be dynamically loaded from a central place don't require some massive, complex and difficult installation processes with access to clients' machines. Note that Java *per se* doesn't give us this. It is easy, as anyone who has built a large and complex Java application can attest, to build one, which requires literally hundreds of classes to run. Such a monolithic application has the very deployment problem we wish to avoid. HTML had the serendipitous effect of forcing application designers to partition their applications. To repeat, the lesson is that applications should be loaded in coarse-grained chunks.

4 What have we learned from servers?

We use a simple analogy here. The analogy is the corner grocery store. Imagine that there is such a neighborhood store and everyone buys her or his weekly groceries there. Now imagine that all of a sudden, everyone's buying patterns changed. Instead of buying a few days' supplies at a time, they bought a single item at a time. In short, a customer would come in, buy a quart of milk and leave. Then return, buy a stick of butter and leave. Then return, buy a bag of apples and leave. And so on. In short order, there would be huge lines trying to buy their groceries at the store. If each time, the customers were buying their goods using electronic cards or checks, the situation would be even worse. Conversely, suppose they were to panic about an impending inflation, coming in to the store and trying to buy a year's supply of groceries. An entirely new and different set of problems would surface. Happily customers don't behave in either extreme fashion.

It turns out that servers are much like grocery stores. Servers can only handle so many clients at once (check out counters). More clients mean longer queues whether at a grocery store or on a server. Servers cannot scale beyond a certain point (you cannot build checkout counters on the fly), and servers cannot just run more and more processes concurrently. If processes increase in an unbounded fashion, their eventual competition for the CPU increases overheads, and overall performance actually degrades.

The cost of starting a new conversation with each customer (client)–or finishing one–has very real costs. Furthermore, if the transactions are too fine-grained, these costs can dwarf the other costs. Moreover, servers aren't designed to serve up really enormous quantities of data (a year's supply of groceries) in a single transaction. Servers bottleneck, TCP/IP complains, and so on. We also learned that processes shouldn't hang onto state while waiting on clients, or other slow processes. As an example, if the clerk doesn't know the price of an item, it would be unfortunate if the clerk simply stopped until the price were found or went her-/himself to find retrieve the posted price. The whole queue would block. Instead, the clerk should get someone else to find the price *and* start helping the next customer if the estimated delay will be significant. In server-speak this means that the process shouldn't hang onto state. What we learned from servers is that they should conduct coarse-grained, interruptible conversations with their clients.

This is very important. It means that when talking to a medical system, or a bank's payments system, or a purchase order system, the client should exchange information in coarse-grained fashion–chunks large enough to let the client go away and do useful work for a while. Indeed, often the serving process should unload all of its state information to the client, and reload it on return. This allows the distribution of state over all the served clients. Combined with the partitioning of applications into components, we can minimize the state information carried by the client, and in need of recovery by the server. In short, it means that good application design for talking to applications on servers involves relatively few methods that can accept and return complex sets of information such as the complete patient record, or the complete portfolio description, or the complete purchase order. This is not how we normally design classes and their methods. Typically, we design for encapsulation, making every access to every property or element a method invocation.

There is another key lesson to be learned from servers. Server software is designed to provide service efficiently and quickly. This is achieved, in part, by sharing transaction protected resources like the databases that underly systems such as those for payment authorization or airline reservations. Server software typically assumes that it is connected through distributed transactions to these resources. This in turn obliges rapid turn-around. Why? Because while any process is holding an open transaction involving these resources, the latter are locked and unavailable to other clients, perhaps even for reading. Notice that any attempt to ameliorate performance by a broader distribution of server code connects more processors to the transacted resources. If a banking transaction, for example, is debiting an account from one system and crediting an account from another, all inside of a distributed transaction, it must not run over slow or unreliable lines. To do so would increase the risk of locking resources for longer or indefinite periods of time. This means that the code running on the client should not be running within the same transaction as that running on the server. It shares the same information (purchase orders, portfolios, personnel records), but it is code dedicated to some different process such as letting the user view the information. The lesson is, in short, to move (to the client) the information, and possibly some simple portable validation logic for the information, but not the custom code that handles this information on the server. We tend to refer to this kind of distribution as an object to object bridge. Remote procedure call facilities are a traditional mechanism for effecting such bridges. So, to repeat, share data, not code.

To recapitulate, the lessons to be learned are as follows:

- Simplicity and flexibility beat optimization and power when connectivity is key.
- Applications should be partitioned into, and dynamically loaded in, coarse-grained chunks.
- Servers should conduct coarse-grained, interruptible conversations with their clients.
- The distributed interoperation among servers and clients is best supported by the sharing of data, rather than code.

5 Microsoft's opinion

XML is a tailor-made vehicle to move the sort of coarse-grained information (that we have been describing) around the web, whether the information is purchase orders, personnel records, portfolios, or just parameters. It is open, easy, and flexible. It is self-describing through DTDs and (eventually) through XML schemas. When the proposals before the W3C XML Schema Working Group[1] coalesce into a standard, such schemas will support rich extensible datatypes and additional metadata that an application will need in order to understand how to use or interpret data presented to it as an XML document. Virtually any logical view's data can be described and transported using XML. Indeed, we view XML as the *lingua franca* for gluing together the computation, communication and storage components of web-distributed applications.

6 What is left to do to enable this interoperation?

6.1 Enhance XML to support extensible and rich typing and metadata

The logical views described by the data that applications will deliver or accept must include a rich and extensible model for data types. As soon as one tries to use XML, even for something really simple like transporting the arguments that an RPC implies, or transporting a simple set of rows from a relational database, one discovers the need for data types. The recipient needs both to be able to discover the data types, and to be able to reliably interpret them as types appropriate to each programming language. Furthermore, the implied negotiation process will often involve the conveyance of other metadata important for the recipient. For example, if one transmits task descriptions as part of a project description, one may have a default Java class or COM class that should be materialized from this information. Similarly, a middleware search engine may want to get enough metadata about the elements to know which should be shown in a summary view, and which elements should be used as links to related information–and how. In short, the amount of ancillary information that might be desired for XML data is essentially unlimited. This fact argues for a the extensibility of the XML mechanisms for describing XML document instances. While the current XML DTDs *do* allow for describing multiple classes of document instances, the specification language for DTDs is not itself extensible. It is this sort of meta-extensibility that *is* provided by XML-Data, and *will* be provided by the eventual XML Schema standard.[2]

6.2 Describe the schemas

We believe that robust, scalable web-distributed applications will be built around a message passing paradigm. In order for us to enable the interoperation and graceful evolution of these applications, there will need to be explicit descriptions of the messages interchanged. Indeed, we can think these messages as being instances of various languages, where the grammars for those languages are given as XML schemas.

In order to motivate our ideas about schemas and schema extensibility, we ask the reader to imagine a scenario in which a customer is trying to search for providers of used books. The customer has her/his heart dearly set upon some out of print tome by Thomas Carlyle, and wants to know which sites have this book. An eager entrepreneur has decided to deliver such a service, and has managed to convince many used book providers, and even some online book retailers, to publish their inventories in a form specified by a particular XML schema. Instances of this schema will describe books. It need not even be a particularly good or comprehensive schema for describing books. Remember that sites can support multiple schemas (logical views) against a single implementation.[3] The enterprising entrepreneur has also convinced Yahoo to point to the site for book searching. This, in turn, lends a certain urgency to the book purveyors to actually publish a book description schema. Actually, several other schemas are necessary to this enterprise. What are they?

[1] Microsoft is an active participant in the Working Group, having contributed many submissions both formal and informal.

[2] There are three layers to be considered here. There is an XML document instance, an XML schema for that document instance, and a schema (the *meta-schema*) for describing XML schemas. It is commonly agreed in the XML Schema Working Group that both the schemas *and* the meta-schema should themselves XML document instances!

[3] The fact that book descriptions are explicitly schematized means that books descriptions can be marshalled according a schema in the request.

First, of course, there must be a schema for the book descriptions themselves, and most likely a schema for describing the book selling capabilities of sites. Instances of the latter schema would describe a site's physical ability to deliver goods and take payments. Today's XML DTD mechanism is sufficient for the required schematization.

The next schema to be considered is a schema for the services offered. Instances of such schemas (catalogues) provide a way for a search engine to quickly and efficiently determine for each site whether that site has any URL that accesses the book selling services. In addition to the catalogue schema, some protocol or convention will be required for launching the search and returning results. The catalogue schema is one that enumerates which URLs return which book schemas (if any). The protocol or convention is either a magic URL or an HTTP header or verb. This is used to make it clear to the site that the desired information is the list of resources with the filtering restrictions, if any (e.g. which resources support the book schema), that are being passed in through a particular XML schema. Thus we are likely to need yet other XML schemas, one to describe a filter, and one to present a logical view of a site in terms of URLs and schema that URLs can return. Let's call this latter one "Site description" and the former one "Filters" for the moment.

Third, let's assume that the search engine has a way to now discover which URLs return book schemas. Now the search engine may do one of two things:

1. Pull down the inventory and merge it into a database, maintained by the search engine, and tracking which sites have which books. In this case the search engine would like some additional information. It would like to know if the book-purveying site would let the search engine "subscribe" to this URL. In this case, the search engine is likely to want to "subscribe" to the site. This would mean that, from time to time, the site would be told that the inventory on hand has changed. The site would then either ask for the entire inventory over again or ask for essentially a "difference" to the inventory already on file. If the site did the latter, we would need yet another XML schema to describe such difference messages. We'll call this schema "Updates". Presumably, the search engine would discover whether the book- purveying site was willing to accept subscriptions to the URL through the same "Site description" schema that described the URL in the first place.

2. Remember that the search engine site is tracking books. Consequently, the search site would like to find out some additional information. It would like to ask whether the book purveyor site was willing to accept "Filter" requests (as specified by an appropriate schema) asking for filtered sets of these books. Then, if a request for books came in, the search engine would ship such a request directly to the book-purveying site using the "Filter" schema. This model is slower of course, but is a lot easier to implement and less likely to fail.

In either case, as we hope this example shows, "published" XML schemas for describing sites, for asking for filtered subsets and for sending changes are likely to be extraordinarily useful.

However, this model may sometimes be too simplistic. Let's imagine that the site isn't willing to support a general query language, no matter how limited. What it *is* willing to do is expose certain URLs which, when sent the appropriate parameters of Title or Author, return a list of books for those Titles or Authors or both. Notice that this is basically a remote procedure call. Now, even in this case, the search engine would want to send the parameters to the book-purveying site in a simple, easy to engineer manner. If the "Site description" schema assumed above describes the desired shape of the parameters of the "method" and there is a standard schema for marshalling parameters for RPCs, or the "Site description" schema describes the form of the search request, then the search engine knows what XML to send. Ideally, the marshalling schema for RPC will also describe how synchronously the results may be returned and by what mechanism to allow for delayed return. Note that all of the schemas mentioned thus far are well within the purview of facilities already specified in the working draft of the XML Schemas Working Group.

There is another standard language (amenable to XML schematization) that shows great promise for our imagined application. This is XSL or Extensible Stylesheet Language. The search engine we have posited has now developed lists of books. But there is another challenge that faces the search engine: how does it handle the case where different sites support different book schemas? Suppose that there are several competing schemas, or even different releases of an evolving schema. The search engine would like to note them all, and be able to convert between them into whichever common one it prefers. XSL provides mechanism for specifying and performing such transformations.

Similarly, the client receiving the results concerning book availability has a challenge. How does the client dynamically generate the HTML that would be required to display the book list in which vendors might want to include as links back to their sites, or the little blurbs about their site, and so on? The client could, of course, take direct advantage of DHTML and JavaScript, or Java, or any COM language to solve this problem, writing either script or a component to custom generate the appropriate DHTML from the data. The client could also use the data binding features of DHTML. But it would be handy to have a standard way to describe how to build the HTML from the XML, and a standard component (converter) that knew how to execute such instructions. Again, XSL can provide such a transformation.

Lastly, the site vending books has a problem. How does it map its book information, undoubtedly stored as a relational database, into the appropriate XML logical view? Interestingly, the XML schema for the view could itself contain suitable metadata to help compute the answer to this question. For true interoperability such a metadata decoration of an XML schema should be standardized. Even so, undoubtedly a "plan" would be computed for actually building the appropriate XML logical view and it would be an instance of an XML schema, call it the "Database conversion". Also required would be a general converter which given such a schema, would actually talk to the database, submit the requisite SQL, and build the appropriate XML. This model would be extremely useful on any server mapping relational data to HTML clients, even sometimes building the HTML directly.

So, to recapitulate, the following XML schemas would be necessary:

- "Site description" schema: Instances of such a schema would describe the "methods" of a site and the "schemas" it returns. Extensions to this model would also be needed to determine for any of these returned schemas, whether the site is also willing to accept the "Filters" or the "Updates" schema.
- "Updates" schema: An XML schema whose instances describe changes which have been made or should be made to an information repository.
- "Filters" schema: An XML schema whose instances describe desired subsets of an XML logical view.
- "RPC" schema: An XML schema whose instances describe signatures of methods supported by a particular RPC interface.

The following schemas and augmentation of schemas would be useful:

- XSL: It would be helpful to pay some attention to ensuring that the part of XSL that is a tree transformation language is sufficiently easy and powerful.
- Standard metadata in the XML schema language to describe mappings of elements to relational stores.
- "Database conversion" schema: A schema whose instances specify the construction of logical XML views from relational databases.

6.3 Ensure that the programming models for XML stay simple and appropriate

Any model for moving information around the web will require that two classes of applications developers be able to access this information, developers using serious structured languages such as Java or C++, and developers using scripting languages such as JavaScript. The requirements for the two classes are not identical. The scripting programmer tends to have neither pointers nor types, and to want to treat XML as a single big tree of nodes which can be navigated using a collection syntax. Something like `root[foos[3][bars[2][sams[1]]]` would mean that the script writer wanted the first node with tagname `sams` within the second node with tagname `bars` within the third node with tagname `foos` within the `root`. By overloading constructs that are basic to most scripting languages, such a model can be surfaced without requiring all XML data emitters to provide and maintain such indexed collections. What this user code would really translate to is intermediate language code that finds the indicated node(s) using simple enumerators. Conversely, the C++ and Java developers will probably happily deal with a lightweight enumeration model, and build their own object models on top of it, rather than depending upon the layers provided by the XML emitter. Indeed, in many cases, these developers may simply want the nodes pushed out to them, as they will be building their own data structures. It is important that we not overburden each implementation with some top-heavy API which is neither fish nor fowl, but rather build the right low-level API on top of which all can build.

What will be required, in addition to simple enumeration, are ways to extract data based upon the types described in the associated XML schemas, ways to make changes to the data or tree, and ways to validate that the changes conform to the schema of a target XML document. Lastly, as XML schemas described above emerge, facilities to process against them automatically will start to be expected as a service layer on top of the base API. The underlying model, however, allows the implementors to support them directly and natively. In other words, while a service layer might be necessary to find some node using simple predicates, a provider might natively implement support for this in a more efficient manner using internal indexes or hash tables or what have you.

6.4 Stores

We are often asked about XML "stores". We believe that it is unlikely that there will be a single XML store. Different stores will serve different purposes. The cheapest store, of course, is the file system. Many standard components are springing up to provide DOM (Document Object Model) API access to XML stored in streams or files. This includes, of course, our own component which we will make available ubiquitously on all Windows platforms, on Unix, and in Java.

Relational stores are superb at exposing multiple distinct logical views on the same data, and by now have very good scaling and transactional characteristics. Typically, mission critical data will live in a relational store. But this doesn't prohibit the existence of "converters" from relational stores to logical views presented as XML document instances. We expect to see explosive growth in the provision of such views. Ultimately, we would expect that the database stores themselves would be able to make these conversions happen, but that in the near term these converters would be part of middleware living on middle tier servers. Elucidating requirements and presenting the likely successful architectures for such XML view services merit a paper in their own right.

What about object-oriented stores? Many such stores have been applied to new purposes unanticipated by their designers/marketers. These systems are being deployed as "staging" stores for XML data as it is cached on the middle tier or the client. Some will undoubtedly turn out to do an excellent job of providing XML caching. What will be critical here, if we want to see interoperability, is that we avoid a profusion of APIs for talking to these caches. The XML DOM standard helps here by describing how to talk to a particular XML document/object, but doesn't handle the larger issues that are likely to be of concern in caches: versioning, transactions, searching, and so on. We hope that the distributed authoring and versioning (DAV) standard will play a major role in driving convergence in this arena. We also expect the object-oriented technology companies to play key roles here.

6.5 Converters architecture

We believe that for the foreseeable future most data will *not* be stored in XML. Indeed, the main thrust of this paper is that XML acts as a convenient mechanism for fostering application integration at the logical view level, not the physical level. Some data will come from relational databases. Some from Teletype feeds. Some from mainframe databases through CICS. Some from SGML. Some from object-oriented databases. Some from mail stores. Much will be synthesized dynamically by objects written by programmers. But if we want a model for interoperation, we will need a standard component model for transforming between any data whatsoever and XML. We call such components *converters*.

A typical middle tier server will have mechanisms for connecting such converters to queues of XML messages. For example, clients may be sending requests for available classes to a university and expecting a specific logical view (as described by an XML schema). The server at the university would dequeue such a request, pass the XML request to the appropriate converter, probably hooked up to the class schedule (a database on the back end), take the resulting XML, and route it back to the requester. The requester/recipient would display the information, review it, and then, perhaps, want to enroll. The requester would send a message described by yet another ("enroll") schema to the middle tier. The middle tier would dequeue this message and quite possibly hand it off to a completely different converter component talking to a backend student database.

We expect to help produce three sorts of converters to and from XML. These sorts are distinguished by the level of granularity at which they operate and carry out conversions.

6.6 The programming object level

This requires a component model abstracting the concept of an XML converter. These converters will enable the rendering of objects as XML. They will do so by being able to pass back and forth between structures described by object class definitions and XML structures described by schemas. Such converters will also need to act a lot like any other XML provider. They will deliver up XML on request. They will support some standard discovery mechanism such as the "Site discovery" schema. They will need to be able to stream results out. We need to make it easy for people to build such components, even using scripting languages.

6.7 The database level

An extraordinarily common case will be mappings between XML logical views and SQL databases. This is sufficiently common that it should be possible to author such transforms without having to write code or even understand the database schema(s). Indeed, as we suggested earlier, it should be possible to simply send a request for a particular schema described using the syntax of the forthcoming XML schema standard.

6.8 The XML level

XSL has been traditionally viewed as a component that took XML in and emitted nicely printed output using whatever print-medium was chosen. We have looked at XSL somewhat differently. Our chosen output medium is DHTML, whether for dynamic interactive output or for producing rich Office documents. However, we don't want to require that everyone who is trying to produce user a interface for XML be a programmer. We believe that it should be easy to produce DHTML from XML.

Furthermore, we believe that there will not be universal agreement about schemas. It is possible, for example, that 3 or 4 or 5 schemas (rather than one) will be used by sites providing information about their book inventories. However, any sensible program accessing or viewing such data will want to pick a single one. This will require a mechanism for converting from one XML schema to another. If sufficient semantic information is available, it will be possible to do this automatically. But frequently, such information will not be available, and a customer will have to describe how to convert. Again, we don't want to limit this to programmers. We believe it should be easy to produce XML from XML.

Hence, we want a standard extensible schema for converting from XML trees to either other XML trees or to DHTML trees. To Microsoft, the interesting part of XSL is the part that helps define how a given XML tree should be translated into a quite different tree.[4] We anticipate standard converters which use XSL instances to

[4]Indeed, the W3C XSL working group has recently aligned itself this way as well, splitting the standard into transformation and rendering components.

decide how to translate from instances of one XML schema into instances of another, or into DHTML.

7 Implications for the web

XML is the basic building block. A suitably rich XML schema language is the required next step. Then, as we have seen, we do need agreement upon some specific XML schemas and we must remember to keep these schemas simple, open, and easy. If we do all this, the power we unleash to the typical developer is incalculable. It becomes possible to build systems for any line of business application, for collaboration, and systems for intelligent information retrieval including for goods and services.

This revolution will do for applications what SQL partially did for databases. It will open up the flood gates because the number of people who can interact with data in any form will increase enormously. Many difficult problems remain to be worked out, including the details of these schemas, the security issues, transactions, and so on. But in the spirit of the web, we have submitted our proposals for most of these pieces to allow us to start and learn, rather than trying to get it perfect and never shipping.

But the revolution involves not just the "how", but also the "what". Well behaved sites will deliver not only complete corpora of data, but filtered sets of data, updates to data (notifications), and requests to update their data (data entry). In all these cases, they will support these services using standard XML schemas. It will open up an entire new business, namely converters with a myriad of specialized XML schemas that make sense for particular types of transformations.

8 Predictions

The requisite schemas will be defined and implemented. The DOM will converge and streamline. Many database storage systems will start to support both the schemas and the DOM. XML will become a widespread solution to interoperable RPC on the net. Microsoft will actively support all of the above.

Tools will emerge to:

- help author the schemas, be they for retrieval, update, or description;
- view, edit, and manage XML;
- define mappings between XML logical views and databases;
- help programmers use all these components and automatically inter-operate with these components.

Server components will emerge to:

- act as stores and managers and queues for XML;
- act as converters between XML and relational databases;
- act as converters between XML instances described by different schemas;
- marshal objects directly from XML.

The programming model for building applications will change to one in which XML is the standard message format used for transmitting data and for RPC. There will be standard ways to manage these messages through queues, and to register events for handling these messages. Components on the server will routinely save their state into XML data-stores virtually transparently.

9 Conclusions

We're only at the very beginning of the web revolution. The most exciting part is still to come. Soon it will become as easy to interact with programs and data all over the net as it currently is to view shared presentation and content.