



BLUE

“Why I Love EJBs”

By Marc Fleury

Blue, Part 1 of the Red, White and Blue Technology Trilogy

Why I love EJBs, an Introduction to Modern Java Middleware.

Java middleware technology is maturing. Enterprise Java Beans (EJBs) were the first systematic introduction java middleware developers had to indirection and meta-programming. The work we at JBoss Group have done so far is building and then deconstructing the requirements of the EJB specification. I will walk you through the services offered by EJB, how to optimize them, where to obtain performance boosts, as well as show you the design patterns behind the container and how we implement them in JBoss. This paper is intended to show you the power and limitations of EJB technology as it stands today.

First of all I must admit that I love EJBs. I am also a closet MicroSoft Windows lover, so my love for simple technology is cross-boundaries. EJBs were the first comprehensive server-side specification for the java freaks. I have grown through their many phases, from the early buzz days of the specification to the latest final touches of the CMP2.1 release. Back in '98, I had been working on the integration of SAP into Java, or rather how to enable the java layers of SAP. I remember running into the early version of the specification while at SUN and thinking "finally a server side java something". I remember the energy. I didn't quite know what the something was but I remember thinking I should try to make the SAP object model fit into the EJB component model. How silly of me. I remember the early versions of a product called WebLogic, back then with an EJB container based on the RMI registry, how primitive. I remember how buggy it was. I remember the SAP people laughing at me and choosing to go with COM technology. I remember pitching Session beans talking to Entity beans to Vlada Matena, the specification lead at Sun and him laughing me out of the room and rejecting the idea on the basis it wasn't what EJBs were designed to do and that Entities should talk directly to the Web. I remember arguing, quite emphatically with Mark Hapner, co-author of the EJB spec, that the object model of SAP could NOT be standardized in the world of EJB. I remember Mark's keen and clear intelligence. Eventually it all lead to the JCA specification (and the session/entity pattern is today beginner stuff). I remember the economics of a standard drawing by Shel Finkelstein, Sun's marketing manager for J2EE, and the clear mathematical explanation of why natural monopolies emerge in integration industries. I use this picture all the time today. I remember that mostly I didn't really know what I was talking about, but that was alright, because neither did they, no-one did, no one could, these were the days of the beginning, of server-side java and the 1999 gold rush, we were all running wild, everything had to be built and we had a virgin field.

I was as excited as a 3 year old in a candy store. My wife and I were expecting our first child. I left Sun to earn three times as much money as a contractor, but soon came to the realization that I was still using my head to build someone else's product. I remember stopping at a traffic light in the Silicon Valley, being pissed off, and thinking that I would start my own EJB container.

So I went on and created JBoss. I remember the excitement over the word "Open Source" and how finally it was acceptable to the general public and maybe even business. I remember thinking java on Linux would be big. I remember saying so on Slashdot, emphatically, I remember someone answering that "he had never thought of that, of Linux as a middleware platform". I remember the mass of people on Slashdot, the heated discussions about the now mostly marginalized Linux on the desktop. I remember server and start working for

real. I remember knowing this would be a 10 year labor of love. I remember working at night, with a lava lamp, I remember not being able to find good techno music in that period. I remember the fun.

Fast forward.... My memories get blurry.

I remember Rickard Oberg, then a grad student in Sweden, giving me a brand new container design, something totally alien, something based on dynamic proxies and interceptors. I remember how logical, magical and lightweight the result was. I remember throwing away my own work and starting from scratch based on his ideas. I proved the point that dynamic proxies generate superior middleware, way superior to compilation-based approaches. Most of the industry still doesn't get it. Academia for once is running ahead with bytecode generation of proxies, we will use that in JB4. I remember the excitement, I remember the energy around 1.0. It was all so real. We were on top of the world. No one knew it then, the energy and I remember thinking it was misguided. I remember how difficult it was to get any kind of message through there, I remember the noise, I remember the lack of vision, was it panic or excitement? I knew from the beginning this would be big.

Today we have generalized this design, recreated perhaps 60% of the GOF pattern book in a blow-your-mind middleware product. We have pushed the limits of what modern middleware technology can do and we are breaking the mold yet again. We are bringing to you, in an open and readable fashion, one of the prettiest and simplest container designs our industry has known. Let's walk through the EJB advantages before we deconstruct them to better reassemble them.

CONTAINER PERFORMANCE: SIGNAL vs NOISE

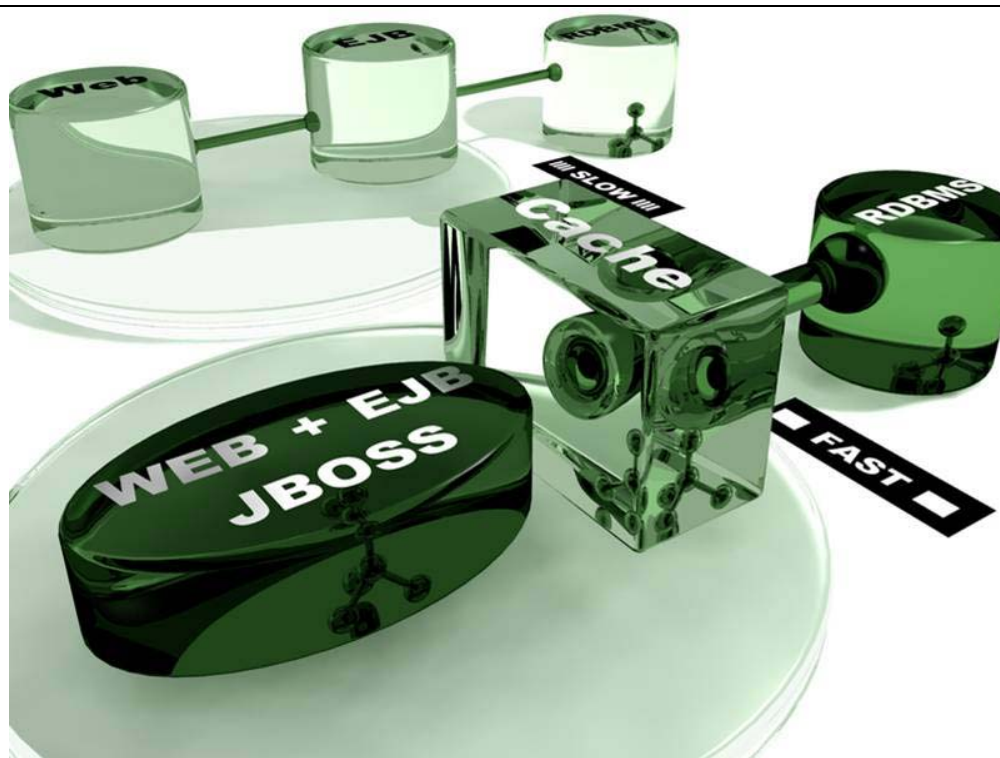
I am a Theoretical Physicist by training. Every physicist knows that when you measure something you want to isolate the "signal from the noise". Sometimes the noise can be the dominant factor and it becomes really tricky to measure the "signal" or what you are interested in. You need finer and finer measurements to go in to the very fast or very small and isolate it from the rest. In container performance the same principal applies. Let me give you the logical picture and the time picture. The logical picture of EJBs is very scary when it comes to wasted time in performance. As a "designer," talking about interceptors and EJB AOP behavior means putting a lot of boxes, representing wasted time, on the whiteboard. The picture of real-time performance looks something very different though. One of the labs we run in our trainings when we are teaching performance gives you the detail on what a container costs time-wise. What the numbers say is that, for most applications, the time spent in the actual container layers is ridiculously small compared to the RPC invocation, i.e. the network invocation of the EJB, and the high amount of time you can waste in a JDBC database driver. The culprit is the old standby, serialization.

In the line up of the "usual suspects" of performance killers, RPC and JDBC stick out like sore thumbs. These thugs are part of a deadly organized-crime family, the Serialization family. The serialization family kills your applications by externalizing data from one process to the other. Instead of working with in memory pointers to the data, the Virtual Machines must spend all its energy and CPU in writing the bytes that describe the data. In that family, the worst of all is possibly SOAP, the Web Services wire protocol, that uses XML to externalize these data structures (a close and just as dangerous cousin is XML/RPC). RMI is no better although not as stupidly brutal. At the end of that family, the nicest gangster is your own TCP writing of the data where you control the schema and exactly what you write to the socket, but let's face it, the fact is that externalizing data structures to byte arrays is, and has been for the past 20 years the culprit of distributed design. A common misconception is that this is an issue with bandwidth. Unfortunately it isn't. The bandwidth is seldom the bottleneck. The bottleneck is Intel. There is at least a factor of 10 in speed between non-serializing designs and serializing designs. Until more CPU is available that problem won't go away. Well, in fact it will never go away,

externalization will always require the CPU to do 10 times more work, because you are doing 10 times more work. When Moore's law catches up, you won't notice the effect anymore. The problem will always be there but you won't notice it since your CPU will be able to do 10 times more work

JDBC is just as greedy if not more. Accessing databases entails serializing data back and forth from the database space to the Virtual Machine memory space. This means serialization and externalization back and forth from the db to the app server. You can clearly say in this case that the "bandwidth" of the network is irrelevant. In this case the "bandwidth" is that of the disks.

So until Intel gives us increased power and databases live in flash disks, we must still look for solutions elsewhere. While flash disk is unlikely to be mainstream technology, an increase of 10 in CPU is certain by the ever-young Moore's law. I argue that even when we hit the 20GHz, you will still crave the increased performance speed we are about to describe since the load and demand for these services will likely increase as their speed and availability increases--the net equivalent of the desktop applications growing in appetite as Intel delivers the goods in speed. So the patterns we are going to see within the scope of EJB will likely apply for the next 10 years of computing. Let's go, EJB are here to save the day with smart RPC optimizations, cache structures and good CMP. .



CACHE IS KING,

The picture shows serialization between web and database, optimizing in JBoss means integrated stacks and presence of caches

CACHE IS KING

EJB's are natural caches. I think that one of the greatest things EJBs have introduced is a formal requirement, at the specification level for distributed middleware cache. This is really powerful, as having the data from the back end systems already mirrored in the java layers, means blazingly fast access times to the data from the web applications. Working with data from cache, meaning from memory, as opposed to retrieving it say from a database or another VM can yield 10x to 50x speed increases. We are not talking about 20% increase or 50% increase or whatever code-clowns like to call optimization, we are talking about orders of magnitude faster. Blow your CPU away with good L2 caches and smart RAM caching. Using caches well is that critical. I am always oscillating between amused and annoyed when people talk about app-server performance. App-server performance is a direct function of how much cache you set up, how much cache hits you have and how long you can keep the validity. How much in-memory work you do equals how little serialization you do $\text{Serialization} = 1/\text{cache-hits}$. You are always up against serialization in JDBC and RPC, local caches of data can minimize the usage of the both, it is that simple. No ifs and buts.

A recent Microsoft study pitches .NET using caches vs J2EE disabling caches, allegedly purposefully. Well done, good for them. Killing caches is the way to kill performance. They proved the point in spades and everyone cries foul... just produce a bench with good caches based on the specification... well that is where it hurts. Unfortunately and for some obscure reason, the specification poorly defines the commit options available to the end user and programmer. This is kind of a shame as a clear definition of their behavior is critical to the speed of the applications and ultimately their scalability.

The spec writers could have standardized so much more than option A and B/C. Just their names, A/B/C tell you how much thought the spec writers put in them, probably none. Also that part of the spec is in an obscure corner--when really it is core material. Also they are broken at a simple level. Let me explain. Option A says you never refresh the cache which is kind of dumb and Option B says you always refresh the cache which is just as dumb, as it means there is no cache. I still don't know what good option C serves really... I mean something as simple as option D in JBoss, that allows you to set an invalidation period for the entries in the cache solved a thorny array of problems for cache management. We also implement advanced cache invalidation patterns like the Seppuku pattern that enables you to directly invalidate entries in the DB, thus creating flexible and sensible architectures for your data plane replication. This level of control is a prerequisite for any real life EJB application. "Real tech not spec" is our new war cry. The fact that you have to dig into some obscure part of the transactional chapter of the spec to even know about the existence of these is a give away. The spec writer for the various versions didn't foresee the importance of caches and the naming convention "A/B/C" (aka dumb/dumber/dumbest) proves that they didn't really think about it at all. The real problem is that there is no assurance of quality and scalability built in the spec, unlike what the marketing material claims. Thankfully, you can move beyond this problem by tuning your JBoss implementation but mostly your application design. Luckily EJB on JBoss comes with a lot of possible optimization patterns at little cost. And "smart Option A's" where you manually control the validity of the cache and assume, until it is invalidated, that the data is good; yields tremendous speed increases for a large array of use cases.

WEB APPLICATIONS COLLOCATION, AVOID DISTRIBUTED DESIGNS

In JBoss, EJBs thus are a robust data cache for the web layers. A typical web application will use servlet/JSP and your favorite portal application to access the EJBs. Since EJBs carry their own optimized caches, we would think that we can work at in-memory speed throughout our stack. Unfortunately, by specification, the EJBs are remote and require the usage of the pass by copy semantic. Serialization is back in full force even as the web layer communicates directly with the EJB container (Tomcat in JBoss is optimized in that way). The specification deals with this in a very "spec heavy" way that requires you to use local interfaces to achieve a very simple result. This is a clear case of the spec gone wild, looping onto its own logic and patching spec-created

problems by adding yet another intrusive layer of complexity in the application design. Local interfaces are a “spec patch” not a first class citizen. In JBoss, we optimize away this RPC layer, transparently and automatically to the EJB programmer. If you are careful with pass-by reference semantics, this delivers on the promise of speed increases. We keep it simple. Collocating the caches and the container thus becomes the right way to do things and is a decision you can take at deployment time. It speeds up your application design. The RMI stack in EJB is superfluous if you don’t need in co-located designs (a bonus of the servlet/EJB integration in JBoss since early EJBoss 1.0 days). Being in cache works, so removing the marshalling in between makes total sense. We do that for you.

Recently I walked into a division of Sabre. They were migrating to JBoss and called me for “arbitrage” on an architectural decision. They had performance problems in their applications and current app-server. When I looked at their architecture something didn’t look right, against all logic, they were ACTIVELY avoiding the collocation of their servlet and EJB tiers. They had a perfectly logical yet silly reason for doing so. They were using BEA and the EJB container was expensive. It, at once and forever, changed my perception of their technical talks. All this crap talk for years, and yet the simplest and most powerful of optimizations isn’t affordable to most clients. How silly is that. This particular client had the price of the license as part of the architectural equation. They had correctly solved that particular equation by avoiding collocation to avoid spiraling license cost but it was the clearly wrong conclusion when we came to technique. Moving to the free JBoss removed this silly parameter from the equation, freed dollars for our services, and gave the choice and options to really optimize the solution to their new equation. Their current design is of course collocated and on JBoss.

2 TIER SERVLET JDBC DESIGN vs WEB-EJB

Any time I hear people doing JDBC straight to the Database (a widespread practice in the early days) they sooner or later but invariably come back with cache problems. The problem is simply stated. If you write a web application, you don’t want each web client generating a request on the database. 1000 requests against the web application server will result in 1000 requests against the back end storage. This is bad, okay. It won’t scale. You want a cache of data to work directly from the same memory space. You want to scale your app-server layer on top of the database. Only if that data is coherently cached does the application server introduce a layer of scalability. Most of these shops end up creating and maintaining their own cache structures. They end up running head first into collocation design problems, cache maintenance problems and they could have saved themselves so much trouble and time by just going with EJB in the first place. Well at least EJB a la JBoss. Entities are natural caches and, at least in JBoss, can be collocated with the web layers at no extra performance (see below on RPC) or dollar cost.

In JBoss, we pay great attention to the locking policies and how access to the cache is synchronized. While working with the lock systems is factored out and you can plug your own locking policy, it is still tricky. A lot of performance and bugs come from the precise coding of the lock policy and the tracking of thread association. We have queued pessimistic locking policies and are adding support for deeply optimistic scenarios. This service is in fact so generic, that generalizing what we did in EJB brings us insights as to how a “cache service” should work for the general purpose AOP framework. We talk about the AOP framework in the RED paper.

CMP 20 persistence, mon amour

Ok, so you understand that caches in memory, alongside with your other logic, are the simplest way to build fast J2EE applications. Now the natural question is “how do we put data in these caches”. Well, today there is very little need to use your own database access code. This is another nail in the coffin of the retrograde servlet to jdbc designs. The new persistence framework, specified in EJB 2.0 is so powerful it should really be a stand-alone spec. We will talk more about that in the RED paper. It is that silver bullet we were looking for in server-side persistence. For those of you who don’t know what it is, CMP stands for “Container Managed Persistence” and this is the age-old promise of automated mapping from the object domain to the database realm. To automate the persistence is simple in simple cases and quite complex for everything else. The first version of the persistence, the 1.1 version, was seriously lacking in various critical aspects. The 2.0 version is however a blockbuster. It comes with Container Managed Relationships, which let the container take care of the integrity of the relationships. The addition of the abstract getters and setters offers another great feature. It is the responsibility of the container to implement these. The nice part is that the container can now follow with great granularity what data you access and what data you write, thereby tracking most of the information it needs to optimize your data access patterns. What this means is that a lot of the optimizations we needed to do by hand in the 1.1 version are now automatically taken care of for you in the container. If you are only reading a bean, we know it, if you are not modifying all the fields we know it, if you are going to load certain data sets for certain methods, we know it since you can give hints to the persistence engine and help it optimize the loading and storing time. One of the interesting things we are seeing with CMP2.0 is that everyone is dropping BMP for CMP. BMP is a fancy word for “JDBC by hand” and was the old way of letting you code your data access layer methods in the body of an EJB. Today, almost no-one uses BMP anymore as the power of CMP is proven and working.

In other words, if the CMP2.0 engine’s applicability goes beyond EJB alone, why couldn’t we imagine a CMP engine working on abstract plain old java objects? We will look at making it the default service for persistence in JBoss. In fact I would argue that CMP2.0 is doing what JDO failed to do, providing a robust and framework-worthy persistence engine for java (once generalized). While it was widely used in designs a year ago, JDO will probably go down in history as the proverbial chicken that crossed the road when the CMP2.0 truck came along.

He who owns the transactional Web owns the Web

I remember meeting the WebLogic founders in ‘98 when they asked me “Is Sun the same f***ed up company it’s always been?” I remember coining the “He who owns the transactional Web, owns the Web” sentence and presenting it to Ali Kutay, the then CEO of WebLogic, while I was applying for a job. I remember thinking, when Ali jotting down the sentence, “hum, that one hit home.” I then remember my surprise when I read a quote from Esther Dyson, a recognized industry pundit, quoting that exact same phrase. A couple of weeks later BEA (the transactional giant) was acquiring WebLogic. “He who owns the transactional web...” was becoming a self-fulfilling prophecy. When BEA told me the news that they couldn’t offer me a job there as they froze hiring, it later motivated me to create JBoss. Incidentally, BEA unsuccessfully tried several times to hire Rickard Oberg, who gave us the JBoss 2.X codebase, which could have been WebLogic...my brain goes in recursive loops at this point, I find it spooky, something to ponder over a good glass of wine.

Anyway, I still believe that the transactions in J2EE are beautiful things. I see so many articles that focus on the in-and-out of XA drivers and 2phi commit protocol and they still miss the beauty of the J2EE transaction. The J2EE transaction is a critical unit of web work. The web transaction accommodates many threads; there is no single thread of control. It defines what scope of your application is working under a transactional flow. That global transaction that can span several components and invocations then drives the local transactions (think

JDBC transaction) by way of XA magic. But really the concept of a web transaction, unlike its JDBC subset, is a very simple and visual one. Most of the transaction specification is truly simple. All the 6 transactional tags mumbo jumbo of EJB really do is help you define where and when you want to start/stop the transaction.

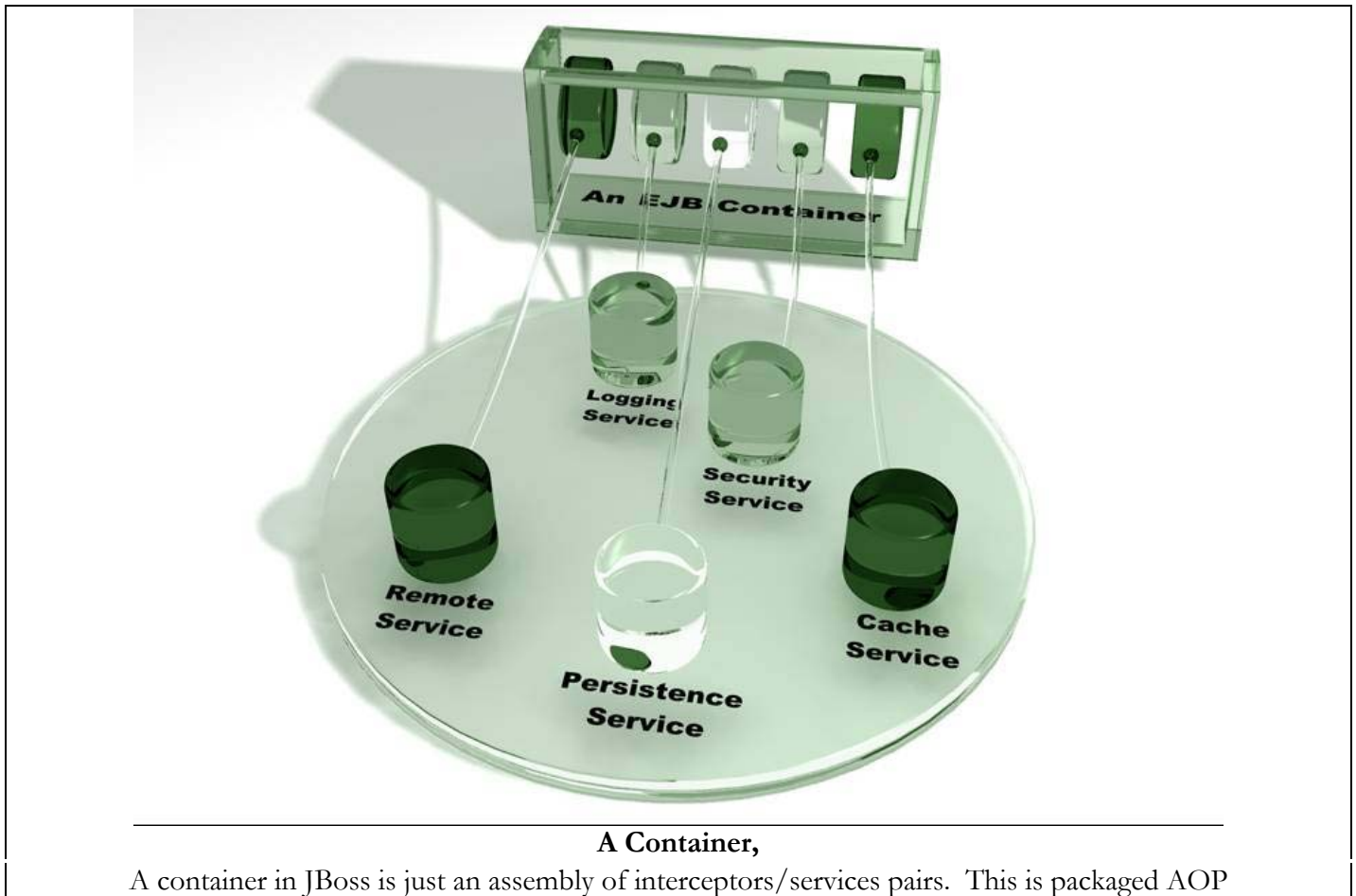
That's it. You can say, in this particular assembly, we start a transaction when we reach this component and we stop it when we are done. This is a very simple and very visual way to view the transaction unit or "critical section" in the web layers. Yet its proper usage is poorly understood and many times people shoot themselves in the foot by abusing the "Supports" tag and enrolling shared components in the scope of the global transaction. This kind of mistake, we quickly find in our JBoss Group Consulting audits. We recommend you use "Requires new" if you want to make sure you isolate your components. But one of the interesting things about the way we handle the transactions in JBoss is that we put a simple "interceptor" in front of your component. Before the call reaches the target we "intercept" and, according to what you define as the behavior for that call, we will start a transaction flow or not. That interceptor isn't tied to EJB. In fact, ANY java method call (EJB or not) that declares the right transactional tags can be outfitted with the transactional EJB-like interceptor and corresponding service and thus expose the same semantics and transactional flow control as the straight EJBs. This is the beauty of AOP assembly taken as a lesson from EJB.

It is my understanding that C# the language from .NET does exactly the same, meaning that you can declare in the body of the code itself. Security behaves in just the same way. We add an interceptor that enables us to control the access patterns.

THE GENERALIZED AOP/EJB CONTAINER

The real crown jewel is the way we have implemented the EJB container. The beauty of EJB, the specification, is that if you are beginning development with server-side java, and specifically with JBoss, you don't have to worry about the individual services. It is done automatically for you. What a great and simple way to start. But that power is also its weakness.

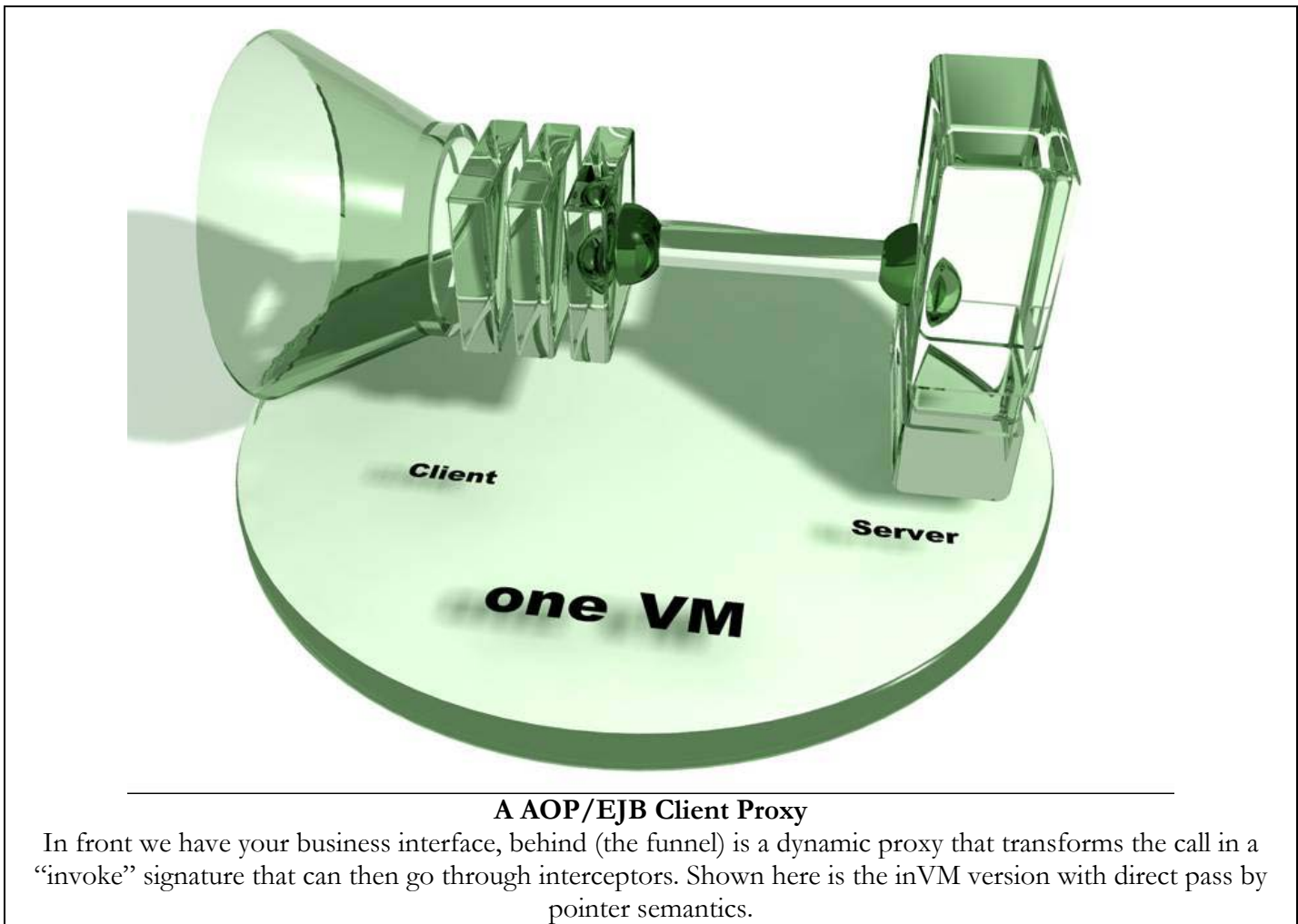
One of the criticisms of the EJB specification is that you need to use the whole package. You may want just the transaction demarcation, just the persistence, just the security model, just the cache, just the distribution, none of the above, all of the above, or a combination thereof...you get the point: it is called modularity. The EJB specification requires you to bundle all the aspects. In JBoss, we implement the container a la carte. As outlined in the transaction part of the paper, we use the programming construct known as "interceptors".



They are simple state machines in the sense of Turin. They take Invocations coming in, they look at the context of the Invocation (for example, the configuration of the transactional attribute for a given method) and apply their logic to the Invocation. In the case of the transactional interceptor this means starting a transaction coming in and stopping the transaction coming out (commit/rollback). It is as simple as that. The key to simplification is isolation of behavior in standalone interceptors. We implement our whole container that way with locking, security, transaction, remote interceptors, clustering, cache and more. This approach has two big advantages. The first one is that we allow for WYSIWYG development of AOP behavior in interceptors. What does this mean? It means that we allow you to develop middleware technology directly in your IDE. [Insert modification of behavior of EJB code.] You can write the body of code in Java and that is what will be running in the runtime. Compare that to the compiler approach to middleware development that our competitors are doing. Think of the hours you waste with EJBC compilation from BEA. This is gone in JBoss. This is a quantum leap in ease of use and development. Not to mention robustness since, to debug this code, we don't need to work inside a black box compiler that generates it, we work directly on the flesh of the code without any indirection. What is interesting is that we allow you to add your own interceptors to the server and client chain. People use this for all kinds of good things, from advanced cache invalidation schemes in the financial industry, for extreme clustering of JBoss instances in the telco world, to advanced security and validation in the military. We give you the hooks to simply specify the interceptor stack you want for a given component.

And then we take this one step further. Having realized the power of the AOP approach with interceptors, we allow you to do this on the client side as well. This is really key to many advanced distributed designs. Let me

explain. The proxy client in the case of JBoss is really a simple and powerful construct. In the front we have the dynamic proxies (see drawing) and they pipe any invocation on your EJB interfaces to a client-side stack of client-side interceptors.



We are unique in the J2EE industry in giving you total control of how these stacks actually look like at runtime. You can add your own interceptor knowing that it will live on the client. The power of this is only limited by your imagination. Just like we have a small interceptor that says in 10 lines of code “if local call locally if remote send the invocation,” thereby adding the local invocation, you can write your own implementation of the “invoke” interface and do any work you want in there. Some clients use it for encrypting critical information on the client proxy and securing the communication with a server-side equivalent interceptor that descrambles the Invocation. Some others view local caches of data where you have client-side session representations that are smart and can answer locally for some requests. One such example is our own EJB interceptor, but you can have a cache of say, stock quotes, on your proxy and it can invalidate every given interval. This essentially gives the end user as developer infinite control over the Invocation and its contents. At the end of the day we extract a map of key-value pairs from the Invocation and send that to whoever it was connected with over whichever protocol (RMI, IIOP, local, JMS, SOAP) is configured in that stack. Your clients can talk with anyone, all configurable from the server. The real beauty of this is that what is done in the proxy, under the user interface definition, is transparent to the client application. Custom security behavior can be added generically to ALL your components transparently to the client code. This is the AOP dream made real for the masses.

And this is really the message I want you to take away from this white paper. Yes EJB were a godsend, yes doing EJB right requires some knowledge but once you know it, the JBoss implementation is an extremely flexible framework. With “beyond EJB” in JBoss, our real EJB on JBoss project, we introduce you easily to one of the most powerful AOP frameworks in the industry. With its complete configurability of plug-ins and interceptor you control, through simple XML files, the exact behavior of the EJB invocations. We give you hooks in the client- and server-side to add needed “real-life” features to your applications. With EJB on JBoss you can become a super-developer, modifying the container AOP behavior to suit your exact needs. This is not customization for the sake of customization, this is used every day for very serious applications in many industries: Real Tech Before Spec. Give it a try, the plain vanilla EJB that is. Come to one of our trainings and learn exactly how to use and tune all the parts, even modify interceptors for your own benefit. Everyone does it, it is simpler than it sounds. In fact, everyone usually finishes the lab on client side interceptors in less than 30 minutes. You will find its simplicity refreshing. You will become a container developer before you know it.

Oh and remember we love you, soon, you will all be walking through walls, not just me, everyone.

Marcf



The TWINS and Papa, sound asleep
Blaise and Leo, 5 weeks old