Introduction to Server Side Emulation

Revision 4 (Final)
4/2/2006 7:26 PM
By Corillian
http://cellframework.sf.net

Welcome to the *Introduction to Server Side Emulation* (SSE). This paper is intended to be used as a guide to starting your first emulator project. If you are having trouble figuring out where to begin - this guide is for you!

Prerequisites

Server side emulation is a complicated task and not for the faint of heart. It is recommended that you are comfortable with at least one programming language. C/C++, Java, and C# are pretty standard within the server side emulation community. A good understanding of hexadecimal and binary is absolutely essential. If you have any problems with the aforementioned subjects I highly recommend you master those skills before you continue. Some familiarity with x86 ASM is recommended.

Crash Course in Server Side Emulation History

When server side emulators first started nobody really thought about recording it. That being said it can be hard to find accurate sources of information. The archives of Smithy's Anvil are probably the best resource on the internet.

The very first server side emulator is typically credited as being UOX from the early days of Ultima Online (UO). Here is an excerpt from Xuri's Guide to UOX3 on the history of UOX:

"*UOX3 stands for Ultima Offline eXperiment 3.*
It is a server emulator for OSI's Ultima Online Server, and the very first version was created by the mysterious figure known as Jaegermeister way back in 1996 or 97. Marcus Rating (Cironian) took over development of UOX, and then created UOX2, followed by UOX3 (current version)."

Shortly after UOX became UOX3 another UO emulator by the name of Grayworld was formed. Grayworld turned into TUS which eventually evolved into what is known as SPHERE today. Soon after, a game by the name of EverQuest (EQ) was released. EQ actually went a while without having an emulator but then a fellow by the name of Beosil cracked the encryption and started the first emulator - EmuQ. This emulator was squashed by Sony's legal team fairly quickly and never had a public release. A year or two later EQ finally had its first emulator release by the guys over at http://hackersquest.com. You could probably say that this was around the time that emulation within the UO community had taken on a frenzy and many UO emulators started popping up (many based off of UOX3). Asheron's Call (AC) also eventually acquired some emulators of its own but as far as I know they never gained widespread popularity. Now we come to the release of Dark Age of Camelot (DAoC). This game went about a year or two before a group that eventually called itself CoAD started an emulator for it. A year after that another emulator known as Dawn of Light (DOL) was started. By this time server side emulation had started to go main-stream. Many more people knew what it was and emulator dev teams started to pop up for most major

Massively Multiplayer Online Role Playing Games (MMORPG).

That obviously wasn't the most detailed history ever, but it gives you an idea of what's been going on for the last 7-8 years. There are probably errors in that so if you have a correction/addition to make please let me know and I'll revise it.

Ok, now for the stuff you actually care about

"How do I start my own emulator?"

I get asked that question all of the time. It's a simple question with a not so simple answer. Before you even can think about starting your own emulator you have to make sure you have the correct tools. An outstanding website with many useful applications is www.sysinternals.com. I highly recommend TCPView, Regmon, and Procmon. There are also a great many other tools that are useful so browse through them. In order to crack the encryption, and sometimes packets, you will need to disassemble your target game's client into machine code. IDA Pro and OllyDbg (OllyDbg is free) are widely considered to be the best option when it comes to disassembling and machine level debugging.

The Community

Your community is your single largest asset. It will make or it will break you. Do not skimp on community management. So why should you pay attention to all those annoying whiners that can never have enough? Emulator development is a massive commitment of time and resources. I cannot think of a single successful emulation project that doesn't measure its existence in years. When you start an emulator you're in it for the long haul. If nobody uses your software then what's the point? Also, nine times out of ten, your community wants to help you in any way it can. It is in your best interest to harness this energy. An excellent way of doing this is releasing your packet sniffer to the masses and providing them with a way to work with you on cracking the packet structures. This helps you out in a big way and it also gets them involved (which keeps people interested and motivated).

Order of Operations

Most Massively Multiplayer Online Game's (MMOG) use a series of applications that give the appearance of being one single game. The order generally consists of a patcher > login app > the actual game client. The one you're going to be most interested in is the game client. Unless you feel like emulating the patch server and the login server as well (though you can usually just write custom protocols for these) it would be a good idea to figure out how to force the game client to start up on its own. By doing this you can also force it to connect to whatever you want to. This is good if you decide to write a proxy sniffer for your packet sniffer.

The Sniffer

The very first thing you should do after you have all the tools you need is write a packet sniffer. Many people will try to cheat and use something like Ethereal. I highly recommend you avoid this temptation and write your own. The reason you do this is because just about every game in existence uses encrypted socket communications. At some point you are going to have to crack that encryption algorithm and put it into your packet sniffer so you can actually understand what the client and server are saying to each other.

Some different ways to write a packet sniffer are the following:

- **Proxy Sniffer:** Tricking the client to connect to your proxy server which then forwards all the data received from the client to the server (and vice versa). As the data passes through your proxy you copy it into a log file. This is also commonly referred to as a "wedge".

- **Hooking Winsock:** Loading a .dll into the client address space and hooking the Winsock API (specifically recv(), send(), sendto(), and recvfrom()).

- **Winpcap:** This is a kernel mode driver for capturing packet data over an Ethernet adapter. I've found this to occasionally not work but if it does it's an excellent solution.

All of the aforementioned methods will accomplish the same goal so choose whichever one you are most comfortable with or best fits your needs. It is in your best interest to put a lot of time into writing a quality packet sniffer. This brings me to one of the most important things in emulation development.

Packet Cracking

Packet sniffers capture packets in the form of blocks of binary data. These are then usually written to a file as hex with a header telling you its source, destination, and the size in bytes. Packet cracking is taking one of these generic chunks of binary and trying to make some sense out of it. This is a very long and tedious process that most people hate. Once you get good at it would be a good idea to solicit the help of your community (release your packet sniffer remember? Maybe provide a way for them to directly put new packet structures into it via scripting or something similar).

Let’s take a look at what we're trying to do here. An example hex dump that I just made up is:

[Protocol: TCP, Direction: Server -> Client, Size: 8 bytes]
0x00 0x06 0x00 0x0a 0x00 0x00 0x00 0x0b .....

Hex dumps are usually formatted in the form of individual bytes. If you do not understand hexadecimal or how to work with binary and binary operators this would be a

good time to go find an article on it as it's out of the scope of this paper. Anyway, how do we determine what this line of binary is? First off, every packet begins with a packet header. Most packet headers generally tend to contain some pretty standard information which is the packet ID and the size of its payload. Packet size and ID are usually formatted as byte's or as 2-byte short's. Generally the first byte or set of bytes is the packet size or its ID and the rest of the header will be things like sequence, checksums, etc... In the above example the first set of bytes is the packet size. We know this because the entire length of binary is the same as the first set of bytes (minus the size of the length value, which is two). We can also see that the packet length value is a 2-byte short. This is because there is a leading 0 before we get to the actual number.

0x00 0x06 = 0x0006 = 6 decimal

Another thing to keep in mind is that the game can purposefully change the byte ordering of its packets. If the above example used a different byte ordering it could look something like this:

[Protocol: TCP, Direction: Server -> Client, Size: 8 bytes]
0x06 0x00 0x0a 0x00 0x0b 0x00 0x00 0x00 .....

Also, a game can only do byte ordering on certain bytes or it can even change it up to try and throw you off. Something to keep in mind is to always pay attention to leading/trailing 0's. That'll give you a clue as to how they're doing byte ordering.

Packet cracking involves a lot of guesswork. Look at the next 6 bytes:

0x00 0x0a 0x00 0x00 0x00 0x0b

Are those six individual bytes? Maybe they're three 2-byte shorts? What about one 4-byte integer and one 2-byte short. It could even be a single 8-byte integer or an 8-byte double. You have no idea just by looking at it. Packet cracking is all about changing something in the game and seeing how it affects a packet. There's a ton of guess work. If pattern recognition is something that comes easily to you then packet cracking will probably be a breeze.

Sometimes you just have to use your head. If you see there's a packet that gets sent every time you move common sense says that is the movement packet. Some of the things that are probably contained in a movement packet are the players x, y, z, and heading. There's a high probability these are 4-byte values and are either integers or floats.

Packet cracking can also be done using a disassembler or debugger. If you can find where the packet handlers are in the game binary this is a superior approach. Reading actual packet structures in assembly consumes far less time than the trial and error of trying to make sense out of raw binary. Keep in mind that this method may tell you the format of packets but not necessarily what each value is used for.

Some things to keep in mind when you're first starting out are:

- Most packets have a header that contains a bare minimum of size and ID.
- Packet headers also can include things like sequence for guaranteeing packets arrive in order, session ids, and checksums.
- Sometimes checksums are appended to the end of a packet instead.
- If you see a string in a packet check if it's null terminated. If not there is probably a value somewhere in the packet that contains the string length. Even if it's null terminated there might be a length value anyway.
- If you see a string where each character has a leading/trailing 0 that means it's probably a Unicode string.
- Test all 4 byte values to see if they're integers or floats. The value of one will probably look like it makes more sense than the other. The same applies to 8-byte values.
- If you see a byte that constantly changes when you change something in-game but the value makes no sense it might be a bit field. Bit fields are commonly associated with flags for things like player appearance/race/etc...

It's easy to get frustrated, bored, and burnt out with packet cracking. I cannot stress enough the importance of soliciting the help of your community. When times get tough just keep it up. Generally packet structures don't change that much so once you crack them you probably won't have to do it again. Lastly, try and find a good article on hexing. That will directly apply to packet cracking.

Encryption

Your sniffing sessions have generated hex dumps that don't contain any string literals such as the players name – what gives? This probably means the data is encrypted. In fact, all modern MMOG's I've encountered use encryption so it's generally good to assume it's there in some form.

Encryption is a huge topic with volumes and volumes of information written on it. I highly recommend you buy a good book on the subject as I could spend the entire rest of this paper talking about it and not even scratch the surface. To sum it up real quick, here's how encryption "might" work with your game. Generally the first one or two packets may not be encrypted. This is because those will probably be the key transfer packets. This is assuming the game doesn't just use a private key symmetric algorithm with the key hard-coded in the client binary. If that's the case then all of it will look like junk straight from the beginning. How do you figure out if that's the case? That is specific to the game you are trying to emulate so all I can say is good luck. "Some" of the encryption

schemes I've seen in MMOG's are the following:

- Public-key encryption (a.k.a. asymmetric encryption) such as RSA using the diffie-hellman key exchange.

- Hybrid algorithm where public key encryption is used to encrypt a private symmetric key which is then used after it has been received. The reason for this is because symmetric algorithms are faster than asymmetric algorithms but are not as secure. An example would be a private RC4 key encrypted by RSA. If I remember correctly DAoC's encryption as of spring 2004 is something similar to this.

- Private-key encryption (a.k.a. symmetric encryption) such as Blowfish (used by UO) and RC4. These are generally much faster though less secure than asymmetric algorithms like RSA. If you are a good reverser and the game doesn't use rotating keys you can usually pull the symmetric key(s) out of the game binary (hence the less security).

- XOR (The easiest to crack, pray for this).

- Sometimes packets will contain hashes to make sure a packet hasn't been tampered with during transmission. Common algorithms for this are CRC32, MD5, and SHA1.

- Any combination of the aforementioned algorithms.

Really the only way to crack encryption is to disassemble the client. This is where IDA Pro will become your best friend. Encryption is the single largest block to a beginning emulator. If you can beat it you're pretty much golden from then on out.

Coding the Actual Emulator

The number one biggest mistake n00b's make is features, features, and features. Your community is going to be screaming for new features and you happily oblige them (community management remember?). This is a very BAD idea. It is important that you go slow and develop a solid platform to build your emulator on. Multi-threaded programming has a bit of a steeper learning curve but it's definitely worth it. Asynchronous socket IO using the system thread pool (on windows) makes for a very scalable and solid network layer. Just don't go crazy with synchronizing everything. You only need to synchronize access to objects that will be concurrently read from/written to. Read about semaphores, critical sections, and mutexes. They all have subtle differences and it's good to know which is good for what situation.

Thoroughly test EVERYTHING at every step. In the beginning you're all going to be pumped by the amount of progress you're making and you're just going to want to see what you can do. There's also a pressure to prove that you have something to show. Do

not sacrifice the integrity of your software because of rushing things. A bug that you've spent a week trying to nail down to no avail is not fun at all. Those are also the kinds of things that'll expedite burnout within the project.

Programming Language Selection

All programming languages have strengths and weaknesses. Things you have to look at when choosing a language are:

- The skill level of your team with various languages
- Target platform
- Power vs. Productivity

A language like C/C++ is great if you want maximum control over your server. It works on multiple platforms and is extremely powerful. However, it also has a high learning curve and has lower productivity than other languages.

Java is an excellent choice if you're extremely concerned about targeting multiple platforms. However, Java is slower than other languages and suffers from a horrible socket library (from an asynchronous IO standpoint). Java will provide you with higher productivity than something like C/C++ and many people are familiar with it.

C# is good if you want high productivity and your target platform is primarily windows. It can be used on multiple platforms due to projects like Mono but it runs best on windows. It has an excellent socket library. C#, through unsafe code and PInvoke, allows for many of the low level things available to C/C++ while writing safe code is similar to Java. Ultimately I would describe C# as the middle ground between C/C++ and Java with a sacrifice to platform interoperability.

Open or Closed Source

This debate is as old as Grayworld vs. UOX3. Most emulators tend to be open source, a trend set by UOX3, though some (i.e. Grayworld) are not. Honestly - unless you're writing your emulator in C/C++ it's pretty much pointless to make your emulator closed source. Anyone who has enough skill to write an emulator from scratch is probably going to have enough skill to disassemble C#'s IL or Java's byte code and reverse engineer "your" server. A classic example of this is RunUO which had some problems with people decompiling its assemblies into near perfect source code (it's now open source). In light of that here are some pros and cons for each.

Open Source:

Pros:

- Allows people to help you by submitting bug fixes and spotting errors in your code.

- People can easily make small changes to the source that'll help them to customize the emulator to their needs.

- Many people don't trust closed source projects and sometimes just because you're open source it can generate interest.

- Free hosting by places like http://www.sourceforge.net.

- Your source is available for everybody to critique and as such constructive criticism can be an invaluable tool.

Cons:

- It's easier for people to steal your code and call it their own or use it with malicious intent.

- People can more easily spot and exploit bugs in your code.

Closed Source:

Pros:

- People cannot steal and rip off your code (as easily). Honestly if you produce a good product no matter who tries to rip what – the quality of your product will stand for itself. Most people automatically attach a negative stigma to an emulator that has been ripped off of what they believe is a well written piece of software.

- People with malicious intent will have a harder time finding exploits.

- People with malicious intent cannot use your code as a basis for writing hacker apps for the live servers.

Cons:

- You cannot leverage the facilities provided by the open source community.

- People are more likely to attach a community unfriendly stigma to you just because you won't share your source.

- You may have a harder time attracting talented developers to help you.

Ultimately in my opinion it's good to open source your emulator but close source some things you may consider sensitive (your packet sniffer, the encryption algorithm, etc...). Just because your emulator is open source it doesn't mean all of its associated tools and every single part of it has to be as well (though if there's no security concern it's nice to

open source as much as you can).

Data Storage

The very first rule of data storage is XML is a poor choice. The only thing it's good for is as configuration files, web page generation, and maybe as an augment to your scripting language. SQL is definitely the way to go. Both MySQL and Microsoft SQL Server 2005 Express are freely available and are excellent solutions to data storage for emulators. Another thing that SQL works well with is clustering. To date no emulators that I'm aware of support clustering but I see that as the next evolution in SSE.

The second rule of data storage is you don't need to store absolutely everything in a DB. In fact - you want to store as little as possible in the DB. This is because in a world of potentially thousands of items, player characters, accounts, and NPC's your DB could grow to be quite large very quickly. Also - the more things you store in your DB the more things there are that may potentially require a schema change down the road. Schema changes can be very problematic when there are serious differences between one version and the next.

A good rule of thumb is to store only account information, player character information, guilds, and anything else that is required to maintain the persistent state of the world. This information is very important and must be updated in real-time. Your entire world can be nuked and people will be sad but will probably continue playing once you rebuild it. If they lose their characters, which translate into hours of work, they are probably going to quit and then find you and kill you. It is absolutely essential that character data is always as secure as it possibly can be. Account information, character data, and guild data need to be shared amongst servers in a clustering (or even multi-function server) setup. This makes them excellent candidates for being stored in a DB.

One thing to keep in mind is the player's position doesn't need to be updated (in the DB) every time it moves. If you do that it's going to be a serious performance bottleneck. Things like stats and character state you should save immediately. Things like current health, mana, and position are still important but it's not the end of the world if you log in next time and you're 2 continents away from where you logged off (at least not as bad as losing 3 hours of leveling). Periodic saves of player position, health, mana, etc... are all that's required.

I would also just like to touch on the topic of password storage real quick. Though most server administrators are honest it’s a good idea not to store passwords in a DB as plain text. I personally like to store passwords as SHA1 hashes or as raw binary encrypted with DES. Many people use the same passwords they use for everything else without any consideration for how the passwords are stored. No reason to tempt an otherwise honest administrator. This also better protects your players in the event your DB is somehow compromised by a hacker.

World Data

World saves generally should be at X interval (or at startup and shutdown or both) and should only contain static data. Static data can be defined as anything that needs to be persistent between server startups. Vendors, quest dependent NPC’s, quest dependent items, spawn points, etc... are all examples of static data. The majority of NPC’s and items are going to be considered dynamic. This means that they exist purely to be killed or interacted with in a generic manner. For dynamic NPC’s try storing spawn points which you can plug into scripts to control the types of monsters spawned and maybe limits on where they can travel. Items are slightly more complicated. Items on the ground should be dynamic. Once a player picks it up and it becomes a part of their inventory that item should become static and stored in the DB (player inventory is important information).

Your world file should be a binary flat file format that is sequentially read from/written to. On world startup your server will just read the entire world into memory. Read/write access to memory is considerably faster than it is to a DB or a file on disk. Since static data generally doesn't change much between world saves, if your server crashes you will more than likely only lose a minimal amount of non-critical information. On world saves you just take everything that's in memory and sequentially write it to disk. The only information about an object that should be stored is anything that's required to maintain its persistent state. This way only the minimum amount of storage will be used. Anything that defines behavior should be put into scripts.

Scripting

Scripting is one of the most important subsystems in an emulator. A quality scripting solution will be what sets you apart from everybody else or what leaves a bad taste in an administrators mouth after trying out your software. There are so many different scripting languages out there you should put a lot of time in seeing which one will work best for you. Download them and try them out. Write little examples of what you think you may need your scripting solution to do and run them through it. You can't spend enough time on choosing the right scripting language.

As I have previously stated - anything that defines behavior in relation to logic or command handling should be put into scripts. You should also put NPC and item definitions in scripts. For example you could write a generic dog script. From that script administrators could see how to create a script for maybe a hell hound. The scripts would contain information about how the entity will look in game and how it will behave towards other entities (we'll define an entity as anything that can interact with its environment). Maybe hell hounds like humans but they hate elves. If a hell hound is hit by a certain type of weapon maybe it takes more or less damage or performs a certain action? These are all things that define behavior and should be put into your scripts. You may also want to expose certain aspects of core functionality, such as web page generation and various server statistics, to your scripts as well.

The following is an example of a hypothetical scripting solution using a pretend scripting language and XML. Notice XML describes attributes while scripts only define behavior.

XML defines data and the interface while scripts define behavior

ScriptInterface.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!--Example of a way to use XML and scripts together-->
<!--This XML file tells the hypothetical emulator how to generate its
script interface-->

<!--Example using NPC's-->
<ScriptInterface>
  <NPCTypes>
    <NPC DisplayName="A Dog">
      <Color>Brown</Color>
      <EventHandlers>
        <OnCreate>dog.script</OnCreate>
        <OnDie>dog.script</OnDie>
        <OnSeeEntity>dog.script</OnSeeEntity>
      </EventHandlers>
    </NPC>
    <!--Example of a Hell Hound inheriting from a "dog"-->
    <NPC DisplayName="A Hell Hound">
      <Color>Red</Color>
      <EventHandlers>
        <OnCreate>dog.script</OnCreate>
        <OnDie>hellhound.script</OnDie>
        <OnSeeEntity>dog.script</OnSeeEntity>
      </EventHandlers>
    </NPC>
  </NPCTypes>

  <!--Another example using items-->
  <ItemTypes>
    <Sword dmg="10" name="A sword">
      <Buffs>
        <StrengthBonus>100</StrengthBonus>
      </Buffs>
    </Sword>
    <LongSword dmg="20" inherits="Sword"/>
    <BroadSword dmg="30" inherits="Sword"/>
  </ItemTypes>
</ScriptInterface>
```

Example of pretend dog script similar in syntax to JavaScript

Dog.script

```
function OnCreate(var dog)
{
    //don't do anything special
}

function OnDie(var dog, var killer)
{
    //don't do anything special
```

```
}

function OnSeeEntity(var dog, var entity)
{
    if(entity.IsPlayer)
    {
        if(entity != dog.Owner)
        {
            dog.Attack(entity);
        }
    }
}
```

Example of pretend hell hound script similar in syntax to JavaScript

HellHound.script

```
function OnDie(var hound, var killer)
{
    //hell hounds do 50 fire damage to their killers
    //when they die
    killer.Damage(DamageType.Fire, 50);
}
```

One last thing I would like to mention regarding scripting is that programming languages are not scripting languages. I, among others, have tried to use languages such as C# for scripting purposes and it is my firm belief this was a poor decision. Most programming languages are far more powerful and have steeper learning curves than anything a well designed scripting solution needs. Another problem is most programming languages have to be compiled into some form of a module. Scripting languages allow you to dynamically recompile or simply execute a single file during runtime which is infinitely more helpful in an environment where application downtime is a big issue. Using C# and DOL as an example please consider the following:

C# like Java uses a garbage collector to manage its memory. In the case of DOL when the server starts up it compiles all of the C# script files into a single script assembly. As the server runs the scripts inevitably allocate objects and hand them off to the server. These objects belong to the script assembly. If you need to recompile one script you have to recompile the whole thing. Since there are still references to objects allocated in the original script assembly running rampant the new assembly has to be loaded alongside it. Further more you have no idea when those objects will be de-allocated by the GC. Maybe you try to beat the system by loading the script assembly in a separate domain and unloading the domain before recompile. The likely outcome in this situation is the death of your application. All of the script allocated objects become invalid references when the GC is forced to collect them. Inevitably something will make a call to one of those references and throw a NullReferenceException. The only other alternative is to keep track of every single object that's allocated which is largely impractical. Thus you must shutdown and restart DOL every time you need to recompile a script. As you can see it'll save you a lot of time and grief to do scripting right the first time around.

Team Management

Team management is one of the hardest and yet most essential parts of emulator development. Your team members are probably going to be from all different nationalities and you're only going to know each other over the internet. Someone who's good at coordinating all these diverse backgrounds into focusing on one thing is worth their weight in gold. However, if you are a non-programmer that's the project lead for an emulator keep this in mind. It is impossible for a non-programmer to effectively manage programmers. If your project leader is someone with tremendous management/people skills but is not a programmer I would recommend also creating a lead programmer who works in tandem with the project leader. If your project lead is a programmer with less than desirable management/people skills then it may be desirable to bring someone who does have those skills as an equal or assistant lead.

Everyone is going to want to join your team to get in on the fun. One of the biggest mistakes startup projects make is:

1. Having too many people on the dev team to begin with.
2. Too many non-programmers.

A good startup size is three to four programmers and maybe one or two non-programmers at most (it's a good idea if they have alternate skills like DB programming and/or web dev). Unskilled people on the dev team are just going to annoy your skilled people and too many people in general are going to cause confusion.

Releases at defined intervals with set feature lists go a long way towards staying organized and focused. Having a public dev meeting after every release is a good way to not only keep the community involved but make sure all of your developers are on the same page. If you let your developers just work on whatever they want without any clear direction you're going to end up with a lot of spaghetti code and many problems later on down the road.

- Don't bloat your dev team with too many people.

- If someone doesn't have the skill level to write an emulator, don't let them join. Instead, you can help them out with their questions and maybe direct their energy in a different direction (such as packet cracking). It's always a good idea to be polite and helpful; this contributes to a happy community.

- If a non-programmer is your project leader it's a good idea to create a lead programmer that works closely with the project leader.

- Have clearly defined goals and tasks.

- Do not be a slave driver; people have lives outside the project. The real world simply takes precedence.

- Keep people informed of the big picture.

- Good communication among team members is essential. A private dev chat room on IRC or something will go a long way.

- Thoroughly document all of your code. If the compiler you're using doesn't support this feature take a look at something like Doxygen.

Community Management

Once again back to the community (that's how important this is). Managing your community requires some different skills than managing your team. A part of community management is advertising and marketing. If nobody knows your emulator exists or how great it is then you aren't going to have a community to manage. Once you've got a couple of releases behind your belt and you've acquired a sizeable community it's now a good idea to provide ways (other than forums) for them to communicate amongst themselves. An IRC chat room is an excellent way to do this. IRC chat also provides you with a way to see who some of the most active members of your community are. To keep these people interested and to alleviate burden on yourself it's a good practice to recruit these people as official technical help for your project. A lot of times these people will have extensive knowledge on the usage end of your software. They generally are more than happy to pass this knowledge onto the rest of the world. Another good way to utilize this resource is by having them write documentation for you. One of the most helpful things you can do for new administrators is provide quality documentation on how to get a server up and running and address common problems.

Regular updates on the progress of your emulator are good at keeping people informed and interested. This seems like a no-brainer but it's something that over time tends to fall by the way-side. Providing ways for people to submit feature requests and see the progress being made on them lets them know that their efforts count. The same thing can be said for a bug tracking system.

Keep your dev team involved with certain aspects of the community such as technical help, bug tracking, and feature requests. If an admin has an actual core dev team member tell him how his bug submission is being fixed it'll let him know that you're taking him seriously. If your dev team becomes too distant from the community it can start to run into "us and them" problems. Many things that seem intuitive to your dev team members may not be for the average user. Keeping your dev team involved with the community will make them more aware of what type of people are using their software.

To sum up community management:

- Advertise your project. Try to be as professional as possible. Just because your software is in a gray area of the law doesn't mean it can't be of professional quality.

- Recruit people that are active in your community to become technical help personnel. This should be something outside of the core dev team but it should work closely with it.

- Get the dev team involved with certain aspects of the community. This is beneficial for both parties.

- Keep people updated of your progress.

- Let people know when their feature requests/bug submissions are being worked on.

- If you reject a feature request, let them know why. Maybe they're missing something or maybe you're missing something?

- Don't skimp on documentation. It is extremely helpful to neophytes. Since programmers typically hate writing documentation - utilize the knowledgeable members of your community when you can.

- Search for organizational tools on the internet. A good bug tracker is http://www.mantisbt.org.

Proliferation of Information

This is a new topic I've decided to touch on starting in Revision 4. Firstly I'd like to say that as an emulator developer you are essentially what is referred to as a reverse engineer.

Definition of Reverse Engineering according to http://www.dictionary.com:

"*<system, product, design> The process of analyzing an existing system to identify its components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. Reverse engineering is usually undertaken in order to redesign the system for better maintainability or to produce a copy of a system without access to the design from which it was originally produced.*

For example, one might take the executable code of a computer program, run it to study how it behaved with different input and then attempt to write a program oneself which behaved identically (or better). An integrated circuit might also be reverse engineered by an unscrupulous company wishing to make unlicensed copies of a popular chip.

(1995-10-06)"

This means, in the traditional sense of the word, that as a reverse engineer you are also a hacker.

7th definition of hacker according to http://www.dictionary.com:

"*7. One who enjoys the intellectual challenge of creatively overcoming or circumventing limitations.*"

As such some things that apply to the hacker community also apply to the emulation community. Sharing information is one of those things. It takes many people hours and hours to fully crack and document the inner workings of something as complicated as an MMO. Nothing is more hurtful to emulation communities than trying to hoard information to boost your own ego. Let's take a look at some examples of why it's actually beneficial to you when you share information:

1. **The tale of UO:** With the open sourcing of UOX when it became version 3 (UOX3) the packet information became public knowledge. Soon after UOX3 rip off's started popping up all over the place. Why was this good? As in the business world competition is good. It is something that helps push people to produce the most innovative and quality emulators. In the end it was RunUO who emerged as the victor. The ones who ultimately benefited from this are the server administrators and the community as a whole.

2. **The tale of CoAD:** I was a cofounder of the CoAD project (which is now dead) and thus I am speaking on this from experience. CoAD was closed source from the get go. We chose this in part because we feared Mythic Entertainment would be more inclined to come after us if we didn't. For a long time CoAD progressed at a crawling pace. Due to our being closed source and views against sharing information CoAD remained the only DAoC emulator for quite some time. What eventually became apparent was not only did our lack of information sharing prevent us from more rapidly cracking packets. It started to give us some sense of being elitist. We became selfish with the information as we wanted to have a monopoly on the DAoC emulator community. If you had accused us of such we all would have vehemently denied it as it was something that was more subconscious. These factors, among others, are what eventually led me to quit CoAD and start DOL.

3. **The tale of DOL:** Dawn of Light was started with the community in mind. It has a forum dedicated to nothing but packet information. With a freely available packet sniffer the project quickly attracted many talented people. Almost the entire DAoC protocol (as it was at that time) was cracked within a year of the projects existence (with most of it being within months). The end result was a rapid gain in popularity for DOL which eventually brought about the demise of CoAD. If DOL hadn't been open source and so free with its information I highly doubt the outcome would've been the same.

4. **The tale of World of Warcraft (WoW):** The WoW emulation community is a perfect example of server side emulation gone wrong. In fact it is, as far as I know, totally unique in what a disaster it has become. My theory on how it became this way is that the original WoW emulators (StormCraft etc…) were all closed source and ripping off the free information from the hard working people at places like blizzhackers.com. I'm not going to say they didn't do any of the packet cracking themselves but what they did do they didn't share so it doesn't really matter. After their demise the emulators that replaced them simply followed precedent. Contrary to the standard set by those who have pioneered SSE (and to their discredit) some WoW emulators decided to try to charge money. That is a direct contradiction to everything SSE stands for (both philosophically

and legally). Considering that every generation of WoW emulators has basically cannibalized the ones that came before it (a couple are even based off of UO and DAoC emulators) they are trying to sell something that technically isn't even theirs. This of course, in my opinion, would help explain why Blizzard has taken such a hard line approach towards WoW emulators. Blizzard is the most aggressive company I have ever seen in terms of killing emulators. I can't say that I entirely blame them. The result of all of this is a community that is severely stifled. There are no prominent WoW emulators and those that do still exist are still trying to keep information that they themselves didn't produce behind close doors. If that information was made freely available then no doubt more quality emulation projects would arise. The only people they are hurting are themselves.

The point I'm trying to slam home with all of this is that by sharing your information you are helping yourself and everybody else. History has shown that most emulators who are friendly with their information have done better than most of those who aren't. Another point to consider, using CoAD as an example, what if Mythic Entertainment had closed us down? Since, at the time, CoAD was the only DAoC emulator available all of that hard work cracking packets would've instantly been erased. If you make your information readily available from the get go and you get shut down then it's all already out there. Once something hits the net it's gone - nobody can stop that.

"You may stop this individual, but you can't stop us all..."
-The Mentor, Hackers Manifesto

In Closing

I hope you find this guide helpful. It is the first time I've written something of this magnitude. If you locate any errors or discrepancies please let me know. Anything you'd like to add to this paper please let me know. You can email me at corillian@users.sourceforge.net. All questions/comments are welcome.

References

Reverse Engineering

- OllyDbg – Outstanding free disassembler/debugger.
- IDA Pro – Industry standard disassembler/debugger.
- www.sysinternals.com – Good source of useful tools that allow you to peer into the inner workings of your system.
- http://community.reverse-engineering.net/ - Forums for those who want to learn the art of reverse engineering.

Programming

- http://www.codeguru.com – Good source of programming tutorials, articles, and help.
- http://www.codeproject.com – Good source of programming tutorials, articles, and help.

Emulation

- http://www.shardwire.org – The best emulation community site since Smithy's Anvil went under.
- http://gamerspace.net – A community of emulator developers.
- http://smithysanvil.com/ - Original source of emulation news. No longer operational and only contains an archive of old news.

Encryption

- http://en.wikipedia.org/wiki/Diffie-Hellman_Key_Exchange - Key exchange.
- http://en.wikipedia.org/wiki/Crc32 - CRC32 hash algorithm.
- http://en.wikipedia.org/wiki/SHA_hash_functions - SHA hash algorithms.
- http://en.wikipedia.org/wiki/MD5 - MD5 hash algorithms.
- http://en.wikipedia.org/wiki/XOR - XOR.
- http://en.wikipedia.org/wiki/RC4 - RC4 encryption algorithm.
- http://en.wikipedia.org/wiki/Blowfish_%28cipher%29 – Blowfish encryption algorithm.
- http://en.wikipedia.org/wiki/Private-key_cryptography - Private-key/symmetric encryption.
- http://en.wikipedia.org/wiki/Asymmetric_key_algorithm - Public-key/asymmetric encryption.
- http://en.wikipedia.org/wiki/Encryption - Encryption.

Special Thanks

I would like to give a special thanks to those involved with proof reading and making suggestions as I write and revise every successive generation of this paper. You are becoming too numerous for me to name. Those involved know who they are and I hope this simple paragraph is adequate enough to express my gratitude.