asss Development Guide – 1.2.0

July 29, 2004

1 Introduction

If you're reading this, you probably already know that asss is a server for the multiplayer game *Subspace*, written mostly in C and Python. This document will try to help you to understand how asss works internally and how to develop for it.

There are three types of things you might want to do with asss: modify the existing source (the stuff in the core distribution), write new modules from scratch in C, and write new modules from scratch in Python. You're welcome to do any of those three things, depending on your goals, but I'd like to encourage people to try to write new modules in Python if possible, and only use C if there's a good reason for it (efficiency concerns, linking with other libraries, etc.). Don't let the fact that you don't know Python discourage you; it's a very easy language to learn. Also don't be discouraged by the current incompleteness of the Python interface to asss. It will improve as users submit requests for things that they need added to it.

2 Building

If you want to build all of asss from scratch, there are a few dependencies you need to be aware of: Python, version 2.2 or greater, Berkeley DB, version 4.0 or greater, and the mysql client libraries (any recent version should be ok). If you're building on a unix system, you'll need to use GNU make.

The basic procedure is to edit the definitions at the top of the provided Makefile to point to the directories where your libraries are installed. After that, running make should build all of asss, which consists of a binary named asss and a bunch of .so files containing the modules. Running make install will copy those binaries to the bin directory one level up.

If you're missing one or more of those libraries, you can still build the remaining parts of asss: If you're missing Python, remove pymod.so from the list of stuff to build (the variable ALL_STUFF). If you're missing mysql, remove database.so. If you're missing Berkeley DB, remove persist.so.

2.1 Building on FreeBSD

FIXME

2.2 Building on Windows

FIXME

3 Basic Architecture

I had several goals when designing `asss`: It should be modular, so that server admins could plug in their own custom functionality in addition to or in place of any part of the server. It should support runtime loading, so functionality could be added, removed, and upgraded without taking down the server. It should be robust and efficient.

Those goals led to a design that might look a little scary at first, but is actually pretty simple if you put a little effort into understanding it. However, there's a lot of indirection, and it can be difficult to understand the control flow in certain places, because of the pervasive use of callbacks. Hopefully this document can provide enough information that anyone can understand how it all works, and more importantly, can figure out how to modify or extend it to do what they want.

The three main pieces of the architecture are modules, interfaces, and callbacks.

3.1 Modules

Almost all of the code in `asss` is part of a module (just about everything except `main.c`, `module.c`, `cmod.c`, and `util.c`). A module is just a piece of code that runs as part of the server. Modules can currently be written in either C or Python.

Some examples of modules are `core`, which manages player logins and other really important bits, `flags`, which manages the flag game, `buy` which provides an implementation of the `?buy` command, `pymod` which allows Python modules to exist, and `persist`, which provides database services for the rest of the server.

Modules written in C have a single entry point function.

Modules by themselves can't do very much. In order to be useful, modules have to talk to other modules. The two main ways for modules to communicate are interfaces and callbacks.

3.2 Interfaces

An interface in `asss` is just a set of function signatures. They're implemented by C structs containing function pointers (and rarely, pointers to other types of C data). Each interface has an identifier (a string, although a C macro is used to hide the actual value of the string), and the identifier contains a version number. If the contents of an interface is changed, the version number should be incremented.

Interfaces are used for two slightly different purposes in `asss`: they are used for exporting functionality from one module to others, and they are used for customizing a specific part of the server's behavior. Both uses used the same set of functions, although in slightly different ways, so you should be aware of the differences.

The module manager (one of the pieces of `asss` that isn't in a module itself) manages interface pointers for the whole server. It has several available operations, which are exposed through an interface of its own:

- A module can register an interface for other modules to use. To do this, it creates a struct and initializes its fields with pointers to the functions it's going to use to implement the interface. (Almost always , this struct will be statically allocated.) A special macro is used to provide the identifier of the interface that this struct is going to implement, and also to provide a unique name for this implementation. Then the `RegInterface` function of the module manager interface is called.

  An interface can be registered globally for the whole server, or registered for a single arena only.

- A module can unregister an interface that it has previously registered, using `UnregInterface`. The same arena pointer that is passed into `RegInterface` should be passed into this function. Note that unregistering an interface can fail! See below about reference counts.

- A module can request a pointer to an implementation of an interface, given the interface identifier, using `GetInterface`.

- A module can request a pointer to a specific implementation of an interface, with `GetInterfaceByName`.

- A module can return a reference to an interface that it acquired with one of the previous two functions, using `ReleaseInterface`.

3.2.1 Reference counts

Implementations of interfaces are reference counted. A module that calls either of the `GetInterface` calls that returns a valid pointer owns a reference to that implementation, and must later return it with `ReleaseInterface`. Calling `UnregInterface` on an interface pointer will fail if there are any outstanding references to that pointer (and it will return the number of references).

3.2.2 Arena-specific interfaces

The functions `RegInterface`, `UnregInterface`, and `GetInterface` all take an optional arena pointer. Interfaces that serve only to export functionality will generally be registered globally for the whole server, and there is only one possible implementation for each of them. To register an interface globally, or to request a globally registered interface, the macro `ALLARENAS` should be passed as the arena pointer.

Interfaces that are used to select among different behaviors might be registered per-arena. Passing a pointer to a valid arena to `RegInterface` makes that interface pointer available only to modules who call `GetInterface` with that arena. If a module calls `GetInterface` with a valid arena pointer, but there is no interface pointer with that id registered for that arena, it will fall back to an interface registered globally with that id, if possible. That allows a module to register a "default" implementation for an interface, and let other modules override it for specific arenas.

3.2.3 Priorities

Another feature available when using the interface system to select among different behaviors is priorities. Priorities should be used when it is expected that multiple implementations of the same interface will be registered globally at the same time. Currently, priorities are used when selecting which authentication implementation to use.

An implementation of an interface may specify a priority (any positive integer) using a variant of the macro used to specify the identifier and implementation name. As long as all implementations of that interface are registered with a priority, `GetInterface` will always return the one with the highest priority (in the absence of priorities, the last one registered will be returned).

Note that to use the priorities feature, *all* implementations of that interface must be registered with priorities.

3.2.4 Example: declaring, using, and defining interfaces

Declaring Here's a sample declaration of an interface, taken from `core.h`:

```
#define I_FREQMAN "freqman-1"

typedef struct Ifreqman {
    INTERFACE_HEAD_DECL
    void (*InitialFreq)(Player *p, int *ship, int *freq);
    void (*ShipChange)(Player *p, int *ship, int *freq);
    void (*FreqChange)(Player *p, int *ship, int *freq);
} Ifreqman;
```

The definition on the first line creates a macro that will be used to refer to the interface identifier (which consists of the string "`freqman`" followed by a version number). By convention, interface id macros are named `I_<something>`, and identifier strings are `<something>-<version>`.

Next, a C typedef is used to create a type for a struct. By convention, struct types start with a capital `I` followed by the interface name in lowercase. The first thing in the struct is a special macro (`INTERFACE_HEAD_DECL`) that sets up a few special fields used internally by the interface manager. The three fields are declared as function pointers using standard C syntax.

Using To call a function in this interface, a module might use code like this (adapted from `core.c`):

```
int freq = 0, ship = player->p_ship;
Ifreqman *fm = mm->GetInterface(I_FREQMAN, player->arena);
if (fm) {
    fm->InitialFreq(player, &ship, &freq);
    mm->ReleaseInterface(fm);
}
```

This code declares a pointer to a freq manager interface, and requests the registered implementation of the freq manager interface for the arena that the player is in. If it finds one, it calls a function in it and then releases the pointer.

The freq manager interface is of the kind used to select among alternate behavior. For interfaces used for exporting functionality, typically a module will call `GetInterface` for all the interfaces it needs when it loads, and then keep the pointers until it unloads, at which point it calls `ReleaseInterface` on all of them.

Defining This is a trivial implementation of the freq manager interface, used by the recorder module to lock all players in spectator mode:

```
local void freqman(Player *p, int *ship, int *freq)
{
    *ship = SPEC;
    *freq = 8025;
}

local struct Ifreqman lockspecfm =
{
    INTERFACE_HEAD_INIT(I_FREQMAN, "fm-lock-spec")
    freqman, freqman, freqman
};
```

First the functions that will implement the interface are defined. In this case, one real function is being used to implement three functions in the interface. Then a static struct is declared to represent the implementation. The first thing in the struct initializer is a macro, analogous to the macro used in the declaration. `INTERFACE_HEAD_INIT` takes two arguments: the first is the interface identifier, and the second is the unique name given to this implementation. Alternately, `INTERFACE_HEAD_INIT_PRI` can be used, which takes a third argument that is the priority.

3.3 Callbacks

Callbacks are somewhat simpler than interfaces, although they share many features. A callback is a single function signature, along with an identifier. Callback identifiers aren't versioned, but they probably should be.

Like interfaces, callbacks are also managed by the module manager. They can be registered globally or for a single arena. Unlike interfaces, many callbacks registered to the same identifier can exist at once, and all are used. The module manager functions dealing with callbacks are:

- To register a callback, use `RegCallback`, which takes a callback id, a function to call, and an arena to register it to. Like interfaces, use `ALLARENAS` to indicate a globally registered callback.
- Use `UnregCallback` to unregister a callback. It should be called with the same arguments as `RegCallback`.

- To query which callbacks are currently registered for an identifier, use `LookupResult`. They will be returned as a list.
- After using the list, use `FreeLookupResult` to return the memory used by the list.

Most of the time, you can use a provided macro to invoke all the callbacks of a certain type, so you won't need to use `LookupResult` and `FreeLookupResult` at all.

3.3.1 Example: declaring, defining, and calling a callback

Declaring Here's how the flag win callback is declared:

```
#define CB_FLAGWIN "flagwin"
typedef void (*FlagWinFunc)(Arena *arena, int freq);
```

There's a macro (the naming convention is to start callback macro names with `CB_`), and a C typedef giving a name to the function signature. All callbacks should return void.

Defining To register a function to be called for this event:

```
local void MyFlagWin(Arena *arena, int freq)
{
    /* ... contents of function ... */
}

/* somewhere in the module entry point */
mm->RegCallback(CB_FLAGWIN, MyFlagWin, ALLARENAS);
```

Calling There is a special macro provided to make calling callbacks easier: `DO_CBS`. To use it, you must provide the callback id, the arena that things are taking place in (or `ALLARENAS` if there is no applicable arena), the C type of the callback functions, and the arguments to pass to each registered function. It looks like:

```
DO_CBS(CB_FLAGWIN, arena, FlagWinFunc, (arena, freq));
```

4 Important data structures

There are several important structures that you'll need to know about to do anything useful with `asss`. This section will describe each of them in detail.

4.1 Player

The `Player` structure is one of the most important in `asss`. There's one of these for each client connected to the server. These structures are created and managed by the `playerdata` module. (The details of when exactly in the connection process a player struct is allocated is covered below, in the section on the player state machine.)

The first part of the player struct, which contains many important fields, is actually in the format of the packet that gets sent to players to inform them about other players. The benefit of using the packet format directly to store those fields is that there's no copying necessary when the packet needs to be sent, as the necessary information is already in the right format.

The format of the player data packet, and then the main player struct, will be given below, and then each field will be covered in detail.

```
struct PlayerData {
    u8 pktype;
    i8 ship;
    u8 acceptaudio;
    char name[20];
    char squad[20];
    i32 killpoints;
    i32 flagpoints;
    i16 pid;
    i16 freq;
    i16 wins;
    i16 losses;
    i16 attachedto;
    i16 flagscarried;
    u8 miscbits;
};

struct Player {
    PlayerData pkt;
#define p_ship pkt.ship
#define p_freq pkt.freq
#define p_attached pkt.attachedto
    int pid, status, type, whenloggedin;
    Arena *arena, *oldarena;
    char name[24], squad[24];
    i16 xres, yres;
    ticks_t connecttime;
    unsigned int ignoreweapons;
    struct PlayerPosition position;
    u32 macid, permid;
    char ipaddr[16];
    const char *connectas;
    struct {
        unsigned authenticated : 1;
        unsigned during_change : 1;
        unsigned want_all_lvz : 1;
        unsigned during_query : 1;
        unsigned no_ship : 1;
```

```
        unsigned no_flags_balls : 1;
        unsigned sent_ppk : 1;
        unsigned see_all_posn : 1;
        unsigned padding1 : 24;
    } flags;
    byte playerextradata[0];
};
```

Details on the specific fields of the player data packet:

pktype The type byte for the player data packet.

ship The ship that the player is in. 0 for Warbird, 8 for spectator.

acceptaudio Whether the player is willing to accept .wav messages.

name The player's name. Note: this field is *not* necessarily null-terminated.

squad The player's squad. Note: this field is *not* necessarily null-terminated.

killpoints, flagpoints Part of the player's score. Note that `asss` doesn't use these fields as the authoritative score, and in the future, they might be unused entirely.

pid An identifier for the player. Pids are used extensively in the game protocol, but not used much internally in the server.

freq The player's frequency.

wins, losses More parts of the score. See notes on killpoints and flagpoints.

attachedto Contains the pid of the player that this player is a turret on.

flagscarried The number of flags that the player is holding. This field isn't guaranteed to be accurate, and is only used to help the client figure out where the flags are when it first enters.

miscbits Currently, this field is used only for specifying whether the player has a King-of-the-Hill crown or not.

Details on the specific fields of the player structure:

pkt This is the player data packet described above.

p_ship This "virtual" field refers to the ship field of pkt.

p_freq This "virtual" field refers to the freq field of pkt.

p_attached This "virtual" field refers to the attachedto field of pkt.

pid The player id of the player. It should always agree with the pid value in pkt.

status The current state of the player. See the description of the player state machine below. State values are named with an initial `S_`.

type The client type of this player. Possible values are `T_UNKNOWN`, `T_FAKE` (a fake player created and managed by the server, used for autoturrets), `T_VIE` (a Subspace 1.34 or 1.35 client), `T_CONT` (a Continuum client), or `T_CHAT` (a client using the chat protocol).

whenloggedin This field is used by the player state machine to make the proper transitions when a player is logging out.

arena A pointer to the arena that this player is in. It may be null if the player isn't in an arena yet, or is between arenas.

oldarena This stores the previous value of arena when arena is set to null. It's used to make sure scores and other persistent information is saved properly when switching arenas or logging out.

name The player's name, guaranteed to be null terminated.

squad The player's squad, guaranteed to be null terminated.

xres, yres The player's screen resolution. Only valid when arena is not null and for standard (`T_VIE` and `T_CONT`) clients.

connecttime The time when the player first connected (in ticks).

position The last known position of the player. This contains a few self-explanatory fields: x, y, xspeed, yspeed, and bounty. It also contains a status field, which is a bitfield of various ship equipment.

macid, permid Various identifying values provided by standard clients.

ipaddr A textual representation of the IP address the client is connected from.

connectas If the player has connected to a virtual server that specifies a default arena name, this will point to that name. Otherwise it will be null.

flags These are a bunch of one-bit flags that are used throughout the server:

- **authenticated** If the player has been authenticated by either a billing server or a password file.
- **during_change** Set when the player has changed freqs or ships, but before he has acknowledged it.
- **want_all_lvz** If the player wants optional .lvz files.
- **during_query** If the player is waiting for db query results.
- **no_ship** If the player's lag is too high to let him be in a ship.
- **no_flags_balls** If the player's lag is too high to let him have flags or balls.
- **sent_ppk** If the player has sent a position packet since entering the arena.
- **see_all_posn** If the player is a bot who wants all position packets.

playerextradata This variable-length array is carved up by the player manager to store per-player data for other modules in the server. See the section on per-player data below.

4.2 Arena

Compared to players, the arena struct is relatively simple. Arenas are often used solely by comparing pointers for equality, although there are several useful fields:

```
struct Arena {
    int status;
    char name[20], basename[20];
    ConfigHandle cfg;
    int specfreq;
    byte arenaextradata[0];
};
```

status Stores the loading/unloading state of the arena. Most arenas will be in `ARENA_RUNNING`.

name This is the arena's actual name, used for displaying to clients, keeping track of non-shared scores, and many other things.

basename This is a name derived from the arena name, but will trailing digits stripped off (for public arenas, whose "name" field contains only digits, the "basename" field contains the word "public"). This is used for keeping track of shared scores and for locating settings for the arena.

cfg This is a handle for the arena's main configuration file. There is only one configuration file loaded by default for each arena, although it may include other files, and modules may load different configuration files themselves.

specfreq This field is a concession to practicality. The "Team:SpectatorFrequency" setting was being queried in several places in different modules, so rather than duplicate work, this setting is provided here for modules to use without querying the configuration file.

arenaextradata Like "playerextradata," this variable-sized array is managed by the arena manager to provide per-arena space for other modules.

4.3 Target

A target is a (sometimes implicit) representation of a set of players. Currently, targets are used as a parameter to command callbacks, to indicate who the command should be applied to, and they are also used as parameters to some of the functions in the `game` interface, to warp or prize some set of players at once.

In a command function, targets can be used by accessing their fields directly, or by using the `TargetToSet` function in the `playerdata` interface to convert the target into a simple list of players.

Targets can be constructed simply by declaring one on the stack and initializing its fields. They can also be dynamically allocated, although this isn't often necessary.

Targets come in several types, some of which use additional data (besides the type itself) to specify the set. The data is kept in a C union, since targets can only be of one type at once.

The most trivial target is of type `T_NONE` and means the empty set of players. A single-player target is of type `T_PLAYER` and the `p` field of the data union points to that player. An arena target (`T_ARENA`) indicates all players in the given arena. A freq target (`T_FREQ`) means all the players on a given team in a given arena. Another simple target, `T_ZONE`, means everyone logged into the server. Finally, an arbitrary set of players can be specified using the `T_LIST` type, which uses the `list` field of the data union.

This is the definition of the target struct:

```
typedef struct {
    enum {
        T_NONE,
        T_PLAYER,
        T_ARENA,
        T_FREQ,
        T_ZONE,
        T_LIST
    } type;
    union {
        Player *p;
        Arena *arena;
        struct { Arena *arena; int freq; } freq;
        LinkedList list;
    } u;
} Target;
```

5 Memory management

Memory management in `asss` is relatively simple. Many sorts of memory in the server, such as the global list of players and arenas, are managed by core modules. Others, such as the links of the linked list library, are handled by the utility library, and a module only has to use the linked list functions.

Sometimes, though, a module will need to allocate memory to store private data in. There are three types of memory a module will want to allocate: some amount of space to store data for each player, space to store data for each arena, and arbitrary chunks of memory for whatever use. Each type will require a different way of allocating memory.

5.1 Per-player data

A module can call the `AllocatePlayerData` function in `playerdata` with a number of bytes to reserve that amount of space for each player. The value returned is a key, which can later be used to access that memory, given a player pointer. (Valid keys are positive integers. If the return value is negative, the allocation failed.) Modules that have used `AllocatePlayerData` must call `FreePlayerData` when they don't need the space anymore (typically during unloading).

To access the data, a macro has been provided: `PPDATA(player, key)` which will return a pointer to the start of the per-player space specified by the key, for the given player.

5.2 Per-arena data

This is just like per-player data, except you will use the functions `AllocateArenaData` and `FreeArenaData` in `arenaman`, and the macro `P_ARENA_DATA(arena, key)` to access the data for a given arena.

5.3 Everything else

To allocate arbitrary chunks of memory, use the functions `amalloc` and `afree`, which work just like standard `malloc` and `free`, except that `amalloc` will never return `NULL` (it will halt the server with a memory error instead), and `afree` can safely be used on null pointers.

Also try to be aware of instances where data can be allocated on the stack, which will generally be more efficient than dynamic allocation. If the size of the data isn't known in advance, the system's `alloca` function can be used. If you need to pass a single-element linked list to a function, one can be constructed on the stack by cheating a little bit with the list abstraction, although you might want to use the lists normally to improve the clarity of your code.

5.4 Typical usage

Sometimes a module needs to store a large amount of data for each player, but it's for a specific game type that's only running in a few arenas, and it will only apply to a small number of the players in the zone. Allocating a large amount of data with `AllocatePlayerData` will waste space in that situation, since that reserves space for every player, not just the ones that need it. There are two possible solutions here. One is to just do it anyway, and waste a bit of memory. The other is to allocate only room for a pointer in the per-player data, and have the pointer be null for players who don't need the data, and point to valid data (allocated with `afree` when the player enters an arena) for players who do need it. Which solution to use depends on several factors, such as how many bytes are being allocated, and what proportion of players are expected to need the data.

In general, if the data is more than 20-24 bytes in length and a significant proportion of players are expected not to need the data, you should consider using a per-player pointer and manually allocating the bulk of the data.

5.5 Internals

Per-player and per-arena data work by allocating a big chunk of space at the end of the player and arena structs, which is divided up between modules to store the data. As an example, let's say `playerdata` allocates 4096 bytes of extra space along with each player struct (the exact amount is configurable). It might provide bytes 128-192 of that space to one module to use for its private data, and then bytes 192-204 to another module. The

offset of the range from the start of the extra data array is the key returned to client modules, and the macro simply adds that offset to the start of the array.

This solution is simple and efficient. The only disadvantage is that the amount of extra data for players, and for arenas, is determined at server bootup, and can't be increased or decreased while the server is running. This may lead to situations where a module can't be loaded at runtime because there isn't enough room in the per-player space left. It might also waste a significant amount of space if 4k is allocated for each player but only 1.5k is used by the loaded of modules. An admin particularly concerned about memory might want to check the amount of per-player and per-arena space used by some desired set of modules, then set the allocated amounts to be slightly more than those (just to be on the safe side).

6 Threading

`asss` is a multithreaded program, and will generally have several threads of execution doing important things at the same time. You don't need to know all the threads and their functions to write a module, but you do have to be aware of concurrency issues in shared data.

The most important shared data are the global lists of players and arenas. The player list, managed by `playerdata`, is protected by a read-write lock, and you must acquire it before iterating through the list (the same lock protects a few other items, like player status values). The arena list is also protected with a read-write lock, managed by `arenaman`.

FIXME: write more here

6.1 How to use threads in a module

First, consider well whether you really need a separate thread. Possible good reasons to use threads in your module are: it *greatly* simplifies the implementation of some aspect of the module, or the module makes unavoidable calls that can potentially block for a very long time.

How long a call needs to block before you should consider using threads depends on the path on which that call is made. If it only blocks for a long time during module load, like `gethostbyname` in the `directory` module, then there's no need for a thread. If it's something that happens relatively often, like writing to a file during on every position packet, as in the `record` module, a thread is probably called for.

If you've decided that you need threads, you can simply go ahead and use any of the pthreads library to create and synchronize your threads. A simple synchronized queue for message passing between threads is part of the utility library, and may be useful for modules that deal with threads.

6.2 Internals

The main thread in `asss` runs in the main loop module, and is used for running timer events. The `net` module has three threads of its own, one for receiving packets from the network and processing unreliable packets, one for processing reliable packets, and one for

sending packets. The logging module has one thread that runs log handlers. The `persist` module uses one thread to do its database work, and `mysql` uses a thread to communicate with the mysql database server. The `record` module uses one thread (dynamically created) for each arena that is either recording or playing back a saved game.

7 Persistent data

The persistent data interface is one of the most confusing parts of `asss`, but the concept behind it is relatively simple, so it shouldn't be hard to use it after a little thought.

The service provided by the `persist` module is persistence of per-player, per-arena, and global data. Some examples of things that might use it are player scores, arena statistics (e.g., kills by ship type), a private staff message board, player inventory (in a RPG-type game), and a shutup timeout.

Data stored by `persist` is opaque binary data. Serialization of the actual data that the module wants to store into a byte stream is the responsibility of the client module. Keeping that data around between invocations of the server and sessions of the player is the responsibility of `persist`.

To use `persist` a client module must provide some information, along with three functions that `persist` will call to manipulate the module's data.

The first choice is whether the persistent data is to be stored per-player or per-arena. Note that really global data (one copy for the whole server) counts as per-arena data.

The second choice is the scope of the data. There are two choices for scope: either there is a single copy of the data for all arenas, or there's a separate copy for each arena. The single copy model is specified by `PERSIST_ALLARENAS`, and the one copy model by `PERSIST_GLOBAL`. Note that either option can be specified for both per-player and per-arena data. Per-player global data means there's one copy of the data for each player (e.g., an inventory in an RPG spanning multiple arenas). Per-player all-arena data means there's one copy for each (player, arena) pair (e.g., regular scores). Per-arena global data is simply global data; there is only one copy. Per-arena all-arena data means there's one copy for each arena (e.g., base win statistics).

Note that you don't have to actually store data for each entity. If you want some per-arena data stored only for a few arenas, simply return an empty piece of memory when queried for the data for an arena to which it doesn't apply.

The third choice is the interval that the data should be stored for. This basically indicates when the data gets reset. There are several intervals defined in the server: "forever," which as its name implies, never gets reset; "per-reset," which is supposed to be something like a score reset (around two weeks); "per-game," which is reset at the end of each flag or ball game (or at the staff's discretion); and "per-map-rotation," which is reset when the map changes.[1]

Historical data for intervals before the current one is saved in the database also, and can be queried by the appropriate tools (see the User Guide section on querying the database).

Finally, you must provide a unique key that will differentiate your data from data stored by other modules. A key is just a 32-bit integer.

[1] This currently doesn't happen automatically when the map changes.

After all that information, you'll need to write three functions (no matter whether your data is per-player or arena and what its scope is).

The `GetData` function (you can name it whatever you want, that's just the name of the pointer in the struct you provide to `persist`) is used to query your module for data to be written to the database. It's called when a player leaves an arena or disconnects from the server, or when an arena is being unloaded, to save the data from that entity before it's gone, and it's also called periodically every few minutes, to make sure the data on disk is relatively recent, in case of a server crash.

When `GetData` is called, the client module should serialize its data into the buffer passed into the function, and then return the length of the serialized data. Returning zero indicates that it has no data to store for this entity.

The `SetData` function is called when a player logs in or enters an arena, or when an arena is created. When called, the client module should deserialize data from the provided buffer into whatever form it will use it in.

`ClearData` is called before `SetData` and can be used to clean up memory from the previous version of the data. When called, the client module should set the relevant data to starting values, as if a player or arena with no previously recorded data is being created. `ClearData` will also be called when an interval ends (immediately after a `GetData` call to get the last version of the data), to clear all data for the new interval.

Finally, you pack up all that information and pointers to your functions in a statically allocated and initialized struct (of type `PlayerPersistentData` or `ArenaPersistentData`), and call `persist->RegPlayerPD` or `persist->RegArenaPD`. The `persist` module will be calling your getters and setters from its own thread, so you should use whatever locking is necessary to ensure correctness.

8 The Python interface

FIXME

9 Misc. internals

9.1 The player state machine

FIXME

9.2 The arena state machine

FIXME

10 Reference

10.1 Source code files

FIXME

10.2 Interfaces

FIXME

10.3 Callbacks

FIXME

10.4 The utility library

FIXME

11 Tutorials

11.1 `log_console`

FIXME

11.2 `logman`

FIXME