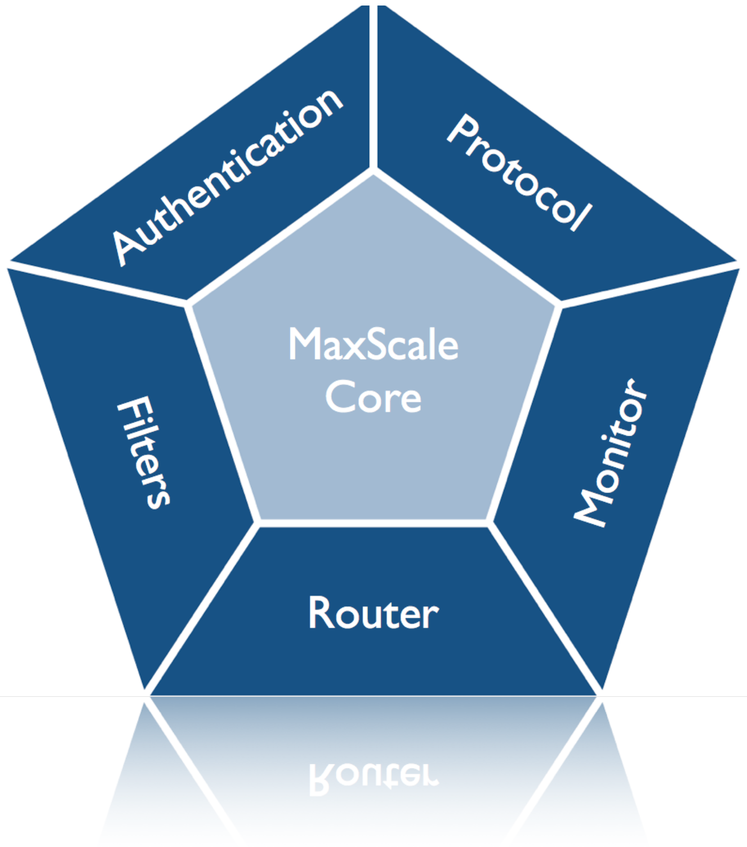
MaxScale Internals

Introduction

Mark Riddoch



Principles & Concepts

Conceptual Overview

- Provide a layer between client applications and the database implementation
- Pluggable Architecture
  - Flexibility
  - Easy extension of capabilities
  - Separation of functionality
- As transparent as possible
  - Ideally no client changes required

Development Practices

- Pseudo object model
  - Per object header file
  - Structure to hold object properties
  - Set of entry points to manipulate object
  - No true encapsulation
- All header files protected against multiple includes
- No header file ordering issues
- All header files include any headers they depend on

Development Practices

- Code documentation via Doxygen
  - Every function should have a comment with @params and @return as a minimum
  - Every file should have a doxygen @file comment with a description of the contents
  - All structures should be commented for doxygen with each field documented
- Test harnesses
  - Every pseudo object should have a test harness

Protocol Plugins

- Two types of protocol plugin
  - Client side (Clients to MaxScale)
  - Backend (MaxScale to Database)

- Responsible for…
  - Connection handshake
  - Data exchange
  - Connection shutdown
  - Encryption
  - Compression

Monitor Plugin

- Monitor backend databases
- Translates backend specifics to generic concepts
- Information provider for core & other modules

Router Plugin

- Two classes of router
  - Connection based - does not look at individual requests
  - Statement based - decides on a per request basis, must look at every request
- Decides which database to send requests or connections to
- Uses status information from the monitor

Filter Plugin

- Sits between the client side protocol and routers
- Filters may be chained together
- Affect requests and optionally responses
- Different filter types
  - Logging
  - Statement modification
  - Statement blocking
  - ‘Utility’

Authentication Plugin

- Allow for non-native authentication mechanism
  - Map generic authentication mechanisms (E.g. LDAP, Kerberos) to database mechanism
  - Map different database models
- Yet to be implemented

Plugin Mechanism

- Fixed set of entry points, same for all plugin types
- Modules return a plugin specific ‘Module Object’
- Model Objects are essentially a set of function pointers
- Plugins are shared library objects, loaded on demand
- Can be implemented in any programming language that can use C calling convention

Plugin Mechanism (contd.)

Module loads, instances and sessions

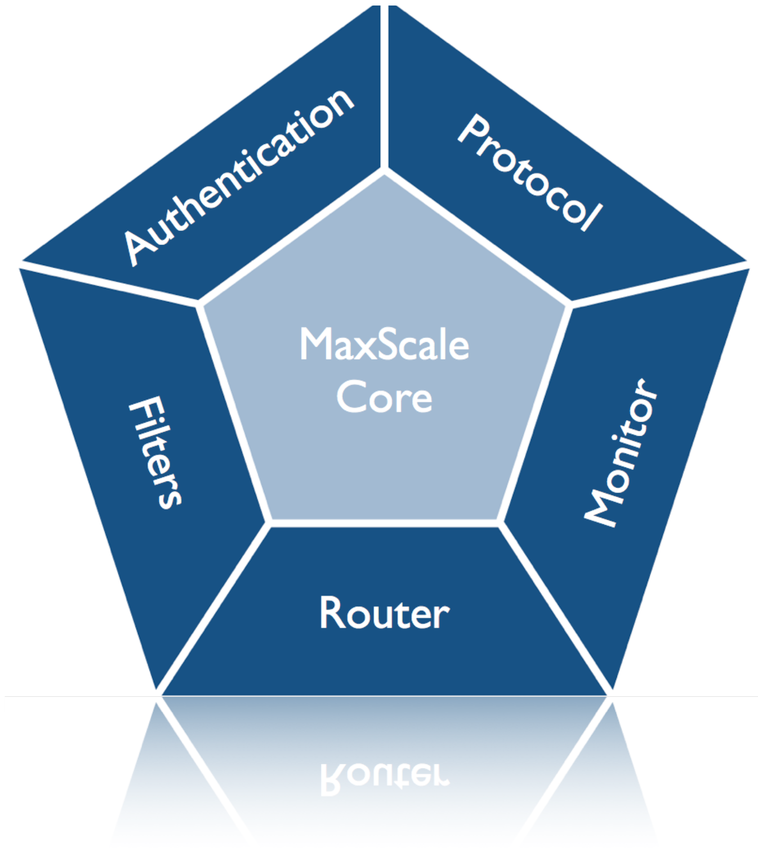
- Each plugin module (.so) is loaded only once
- A new instance of a plugin is created for each use within the configuration file
  - The instance ties the configuration to the plugin
- A new session is created for each client connection that uses a plugin
  - Every session using the same service shares a common instance

Module Common Entry Points

- A set of entry points common to all modules
  - ModuleInit()
    - Called once when the module is first loaded
    - Performs per module initialisation
  - version()
    - Returns a NULL terminated string with the plugin version information
  - GetModuleObject()
    - Returns the plugin type specific module object
    - Module objects are a set of function pointers
- ModInfo structure
  - info static variable with all type and version related data needed to identify plugin

MaxScale Principles

- Object Orientated Design
- Not implemented in OO language
- Separation of responsibilities
  - Not only at the “object” level
  - Plugins should be isolated, self contained and flexible



Event Driven Core

Event Driven Model

- All network I/O is event driven
- Each connection (file descriptor) is given a descriptor control block (DCB)
- The file descriptor and DCB are registered with EPOLL on linux
- Every change of descriptor state causes an EPOLL event, DCB is event data

Event Driven Model - DCB

- Connection state must be passed to event handlers
- DCB is mechanism for communicating connection state between event handlers
- On arrival events are queued for processing by MaxScale threads
- Model maps easily to IO Completion Ports on windows and AIX

Threading Model

- A set of threads for handling I/O events
- A set of utility threads
- No. of I/O threads defined in configuration file
- I/O threads should **never** block
- Best to configure no. of I/O threads to be less than no. of physical cores

Utility Threads

- These threads may block
- Utility threads used for
  - Log manager
  - House Keeper
  - One per instance of monitor plugin

I/O Thread Tips

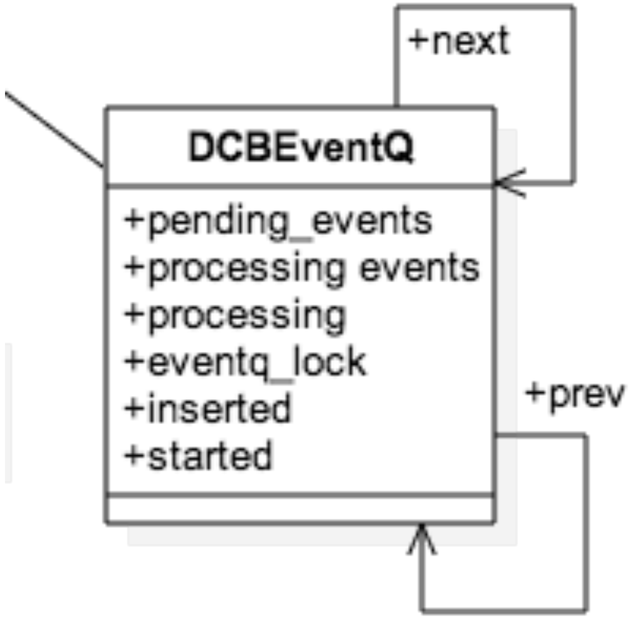
- All network sockets should be non-blocking
- Never wait for a response on a network thread
- Avoid locking objects for extended periods
- Do not usr mutexes that can cause I/O threads to queue on each other

Event Implementation

- File poll.c contains EPOLL implementation of event abstraction
  - Ports to non-Linux system will require platform specific versions of poll.c, e.g. winpoll.c
- Tuning parameters
  - nonblocking_polls - No. of interactions of non-blocking poll before a blocking poll call is made
  - maxwait - Maximum time to do a blocking poll

Event Queue

- Event queue is implemented within the DCB structure
  - DCBEVENTQ structure
    - Doubly linked list
    - Set of pending events
    - Set of processing event
    - processing flag
    - Event queue lock
    - A pair of timestamps
  - When an event is processed the mechanism is
    - Lock dcb->evq structure
    - Copy pending events to processing events
    - Set processing flag
    - Unlock dcb->evq structure

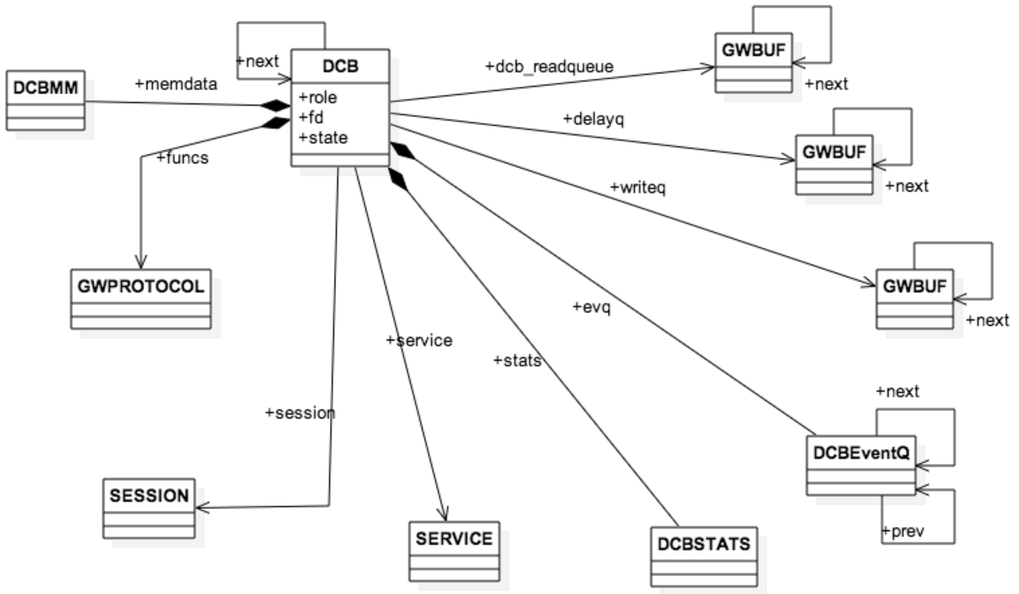


Event Queue (contd.)

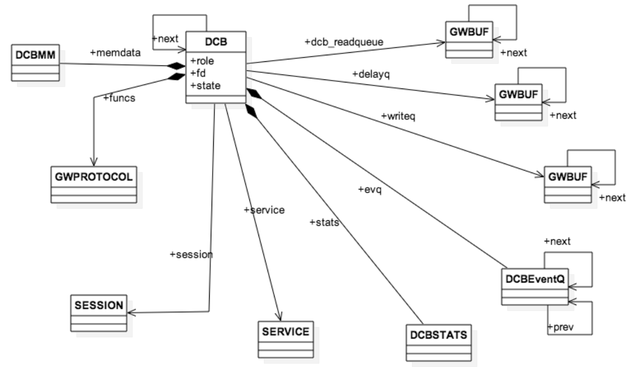
- New events are added in dcb->evq pending events
- If processing flag is set pending events will not be processed until current processing completes
- Threads will not block on dcb->evq lock or wait for processing flag to clear
- MaxAdmin/DebugCLI commands to examine poll statistics and event queue
  - show eventq
  - show poll

Descriptor Control Block

- Descriptor control block is centre of polling
- All events are passed the DCB
- The DCB holds all the connection state directly or indirectly
- DCB's maintain queues of outgoing and incoming data

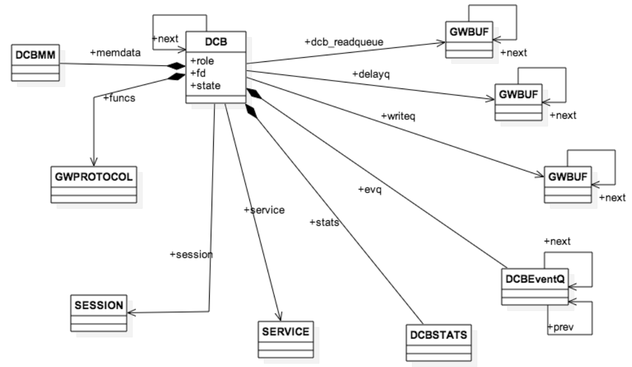


DCB - buffer queues



- DCB->writeq
  - The DCB write queue is used when data is being written to the socket and the socket buffer becomes full.
  - Instead of the write blocking the residual data is added to the DCB writeq
  - The writeq will be flushed when an EPOLL_OUT event is received on the descriptor
- DCB->delayq
  - The delay queue is used to hold requests when there is an outstanding authentication handshake on the connection
- DCB->dcb_readqueue
  - The dcb_readqueue buffers incomplete requests

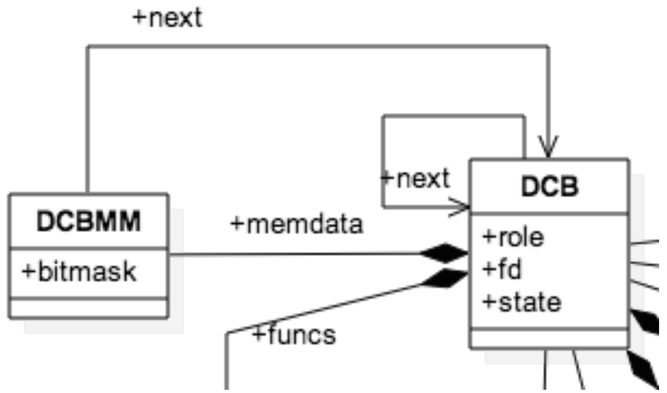
DCB - GWPROTOCOL

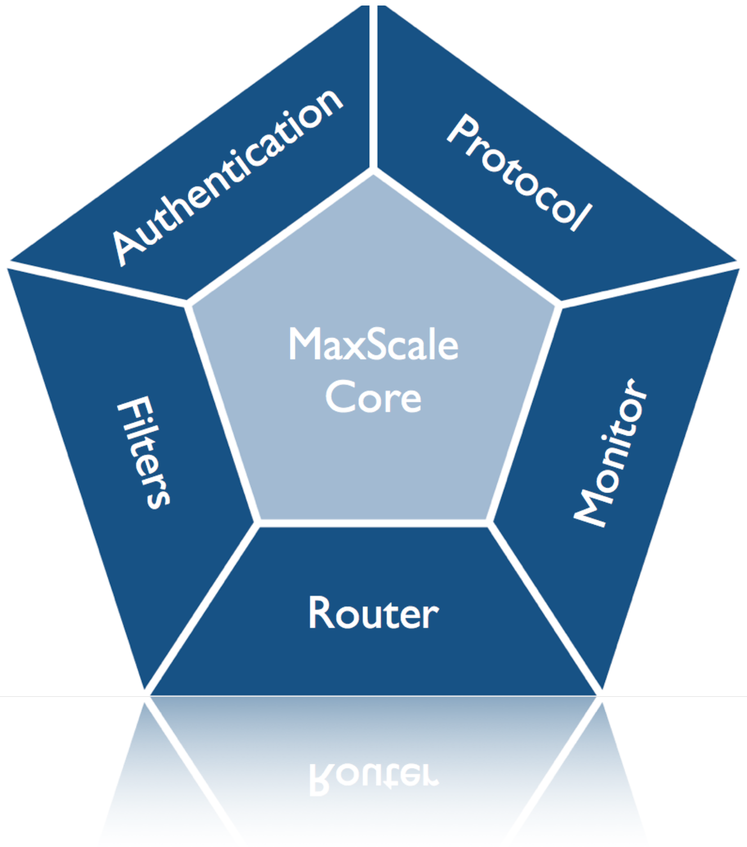


- GWPROTOCOL is the link between the DCB and the protocol plugin
- A set of function pointers that are entry points in the protocol plugin
- These entry points include;
  - read, write, write_ready, error, hangup, accept, connect, close & listen operations
  - These entry points are the links to the protocol specific part of the event handler

DCB - Memory Management

- DCB's may be referenced from multiple I/O threads
- A DCB can not be freed until all threads have finished with it
- DCBMM manages
  - A bitmap - one bit per thread
  - A zombie list
- A DCB is first placed on zombie list
- As each thread completes event processing it checks the zombie list and clears it's bit
- Only when all bits are cleared is the DCB freed





MaxScale Utility “Classes”

Atomic Operations

- Implemented as Intel assembler
- atomic_add - add signed value and return previous value
- Basis for spinlock implementation
- Used for maintaining statistics counters
- Could be replaced with GCC built in function - need to evaluate performance

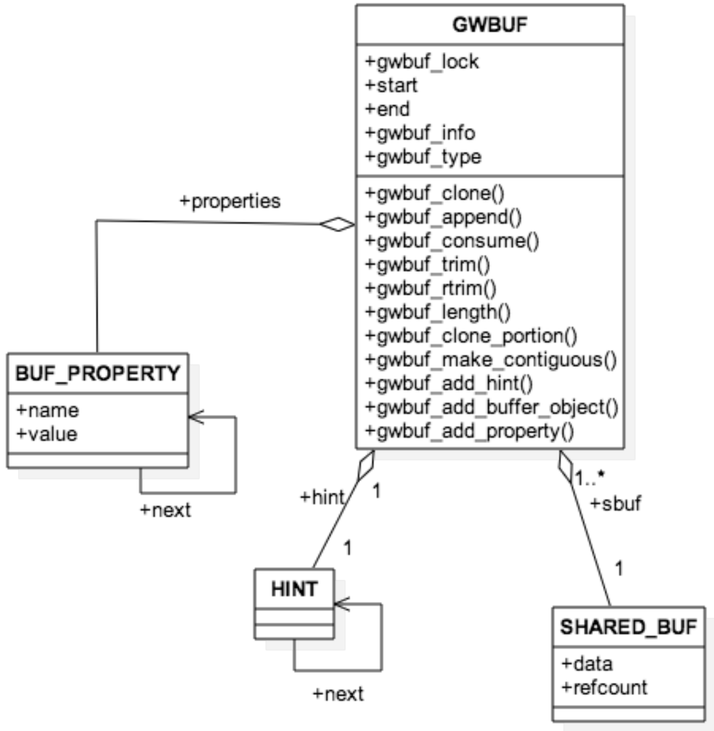
Spinlocks

- Three entry points
  - spinlock_acquire - wait for spinlock to be available and grab lock
  - spinlock_acquire_nowait - grab lock if available, return true if lock was grabbed
  - spinlock_release - release a spinlock we are holding
- All spinlocks must be initialised with spinlock_init or SPINKLOCK_INIT macro
- Spinlocks implement a busy wait for acquisition
  - Spinlocks should not be held for long period
  - Avoid holding spinlocks when making system calls the may cause thread scheduling

Buffer Management

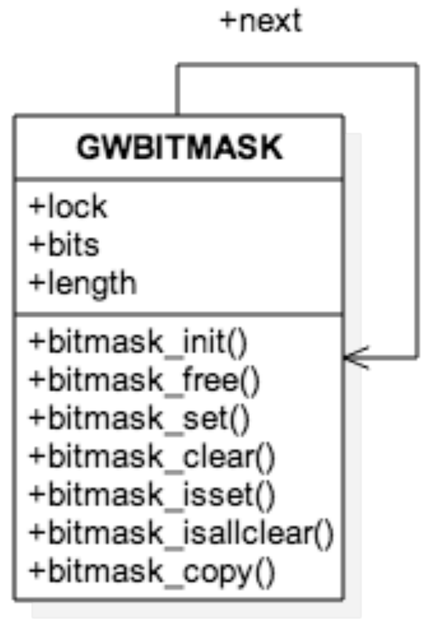
- GWBUF structures designed to allow
  - Sharing buffers between different threads without copies
  - Chopping buffers up without copy
  - Creating lists of buffers for read/write (scatter/gather approach)

- Buffers also allow extra information to be carried with the network data
  - Hints
  - Buffer properties
  - Buffer objects



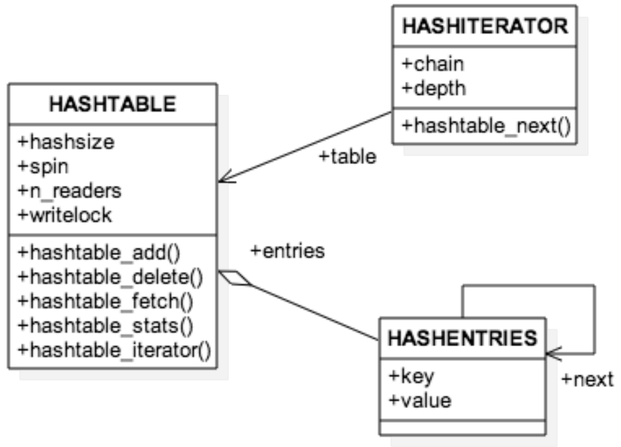
GWBITMASK

- An arbitrarily sized bitmap
- Grows automatically as bits are set and cleared
- Bitmask operations are locked at the bitmask level



HASHTABLE

- Generic hashtable implementation
- Multiple readers, single writer
- Configurable hash function and key/value memory management
- Iterator support for walking hashtable



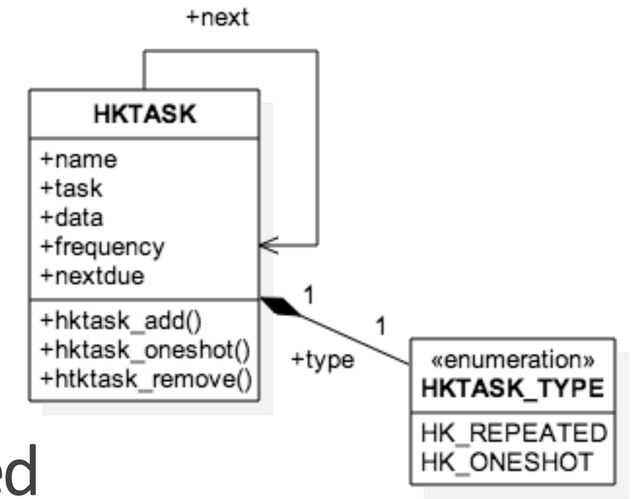
MEMLOG

- A simple interface to allow values to be logged to memory and later flushed to disk
- Simple API to log integer, long, long long and string values
- MEMLogs by be flushed by demand or when defined buffer is full
- Always flushed on shutdown
- Use in time sensitive code

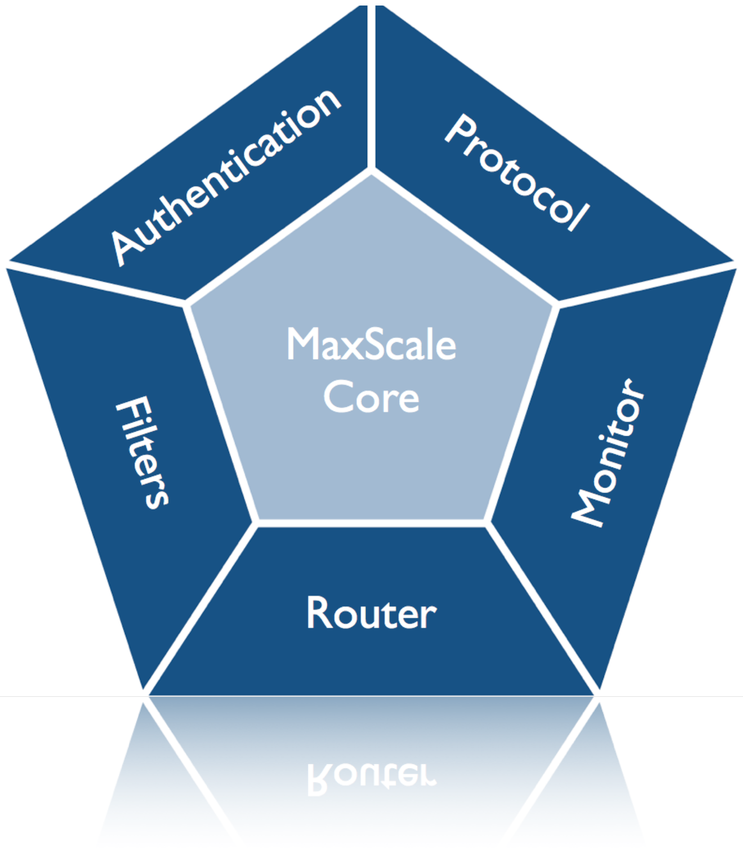
Accurate Time

- RDTSC - low cost accurate time
- Intel processor specific
- Good for profiling
- Returns the processor clock tick value
- Accuracy is related to processor clock frequency
- rdtsc() return an unsigned long long value of current CPU time stamp counter

Housekeeper Thread



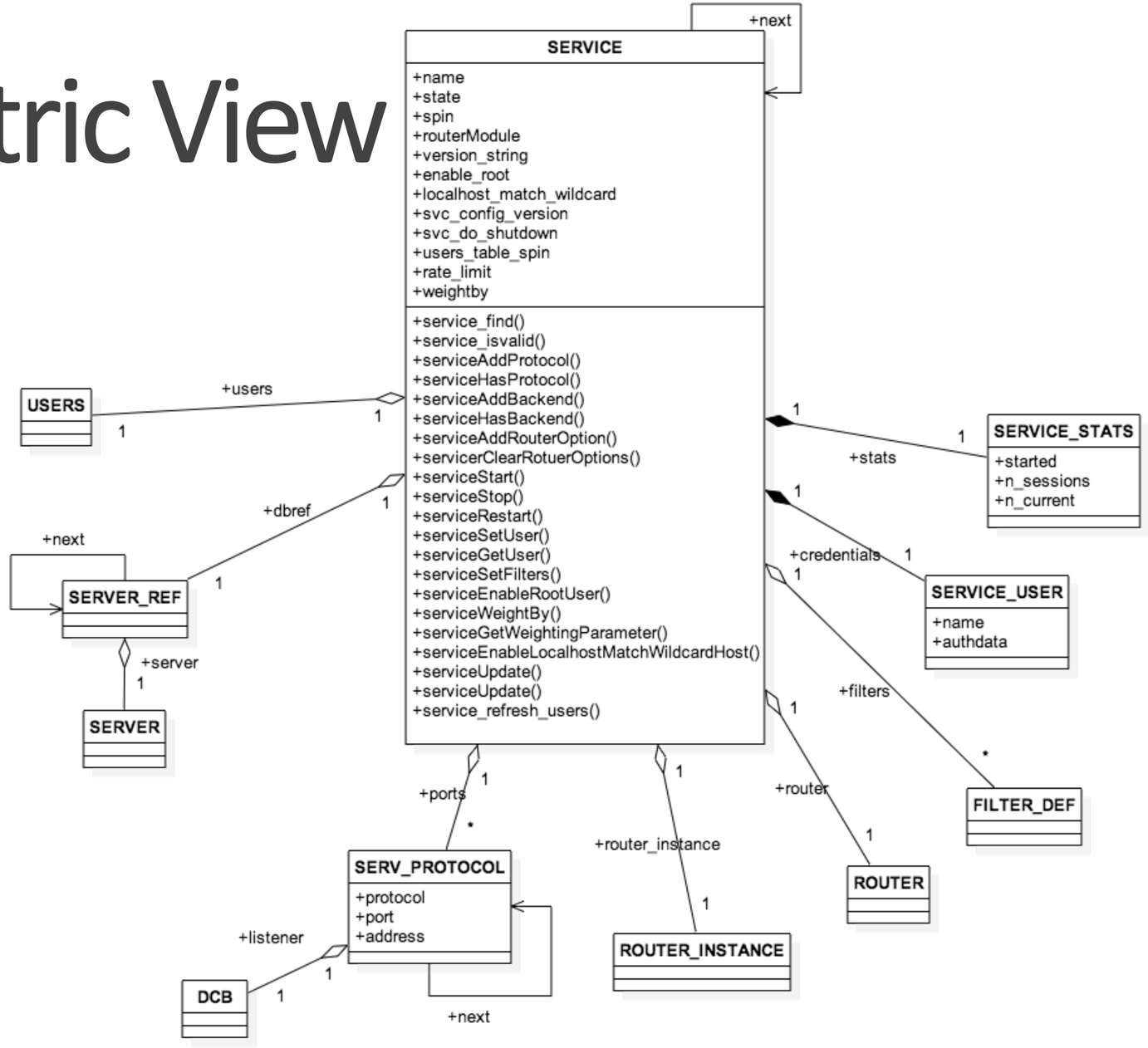
- Housekeeper gives interface that allows timed tasks to be run
- Tasks may be repeated at a given frequency or one-shot delayed
- Tasks consist a function pointer and some user data that is passed to the task
- Housekeeper maintains hkheartbeat counter
  - Incremented every 100ms
  - Globally available cheap source of time
- Implemented with a dedicated thread



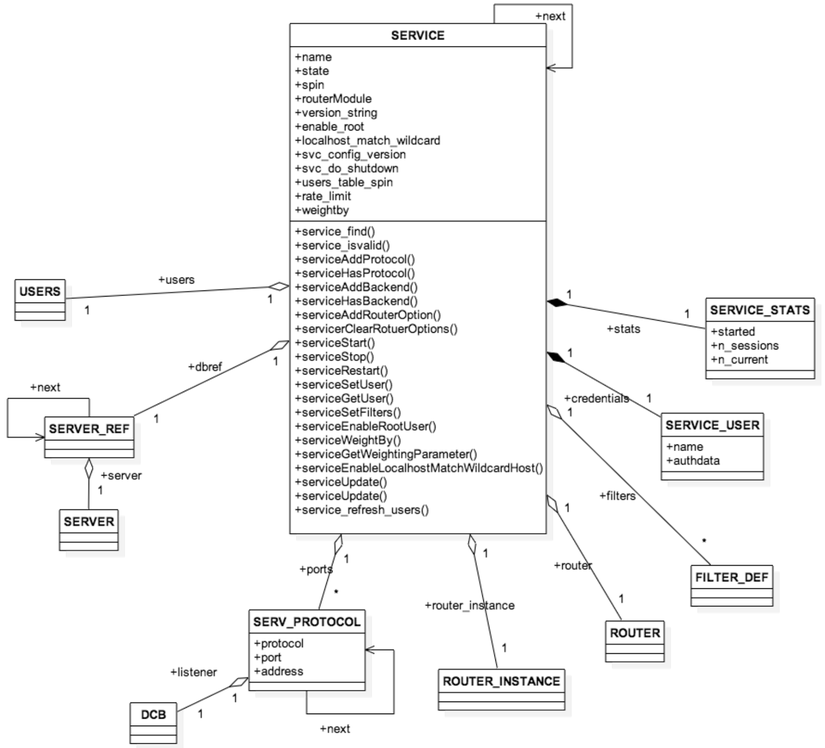
Service View

Service Centric View

- Services are a logical place to start configuration & also to look at internal organisation
- The service represents the static state of MaxScale, i.e. the “potential” for user sessions

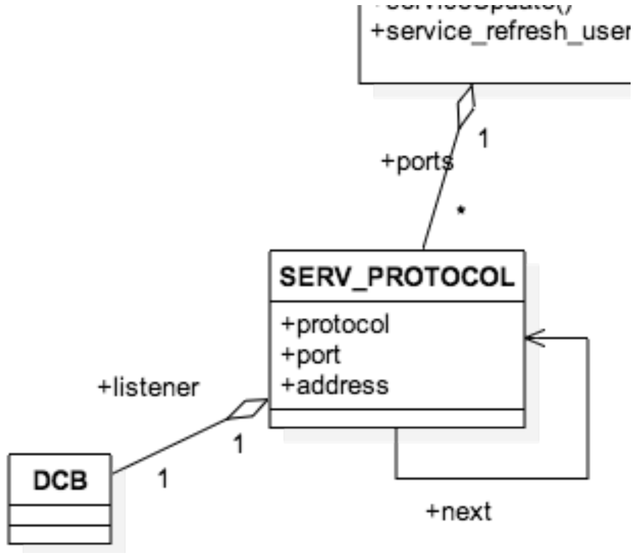


The Role of “Service”



- Services are created as part of configuration load
- Services responsible for loading modules for a service
- Creates instance of plugin with service specific configuration
- The Service ties together all the resources needed to create a session
- The service does not represent an actual session or state, it is a blue print for session creation

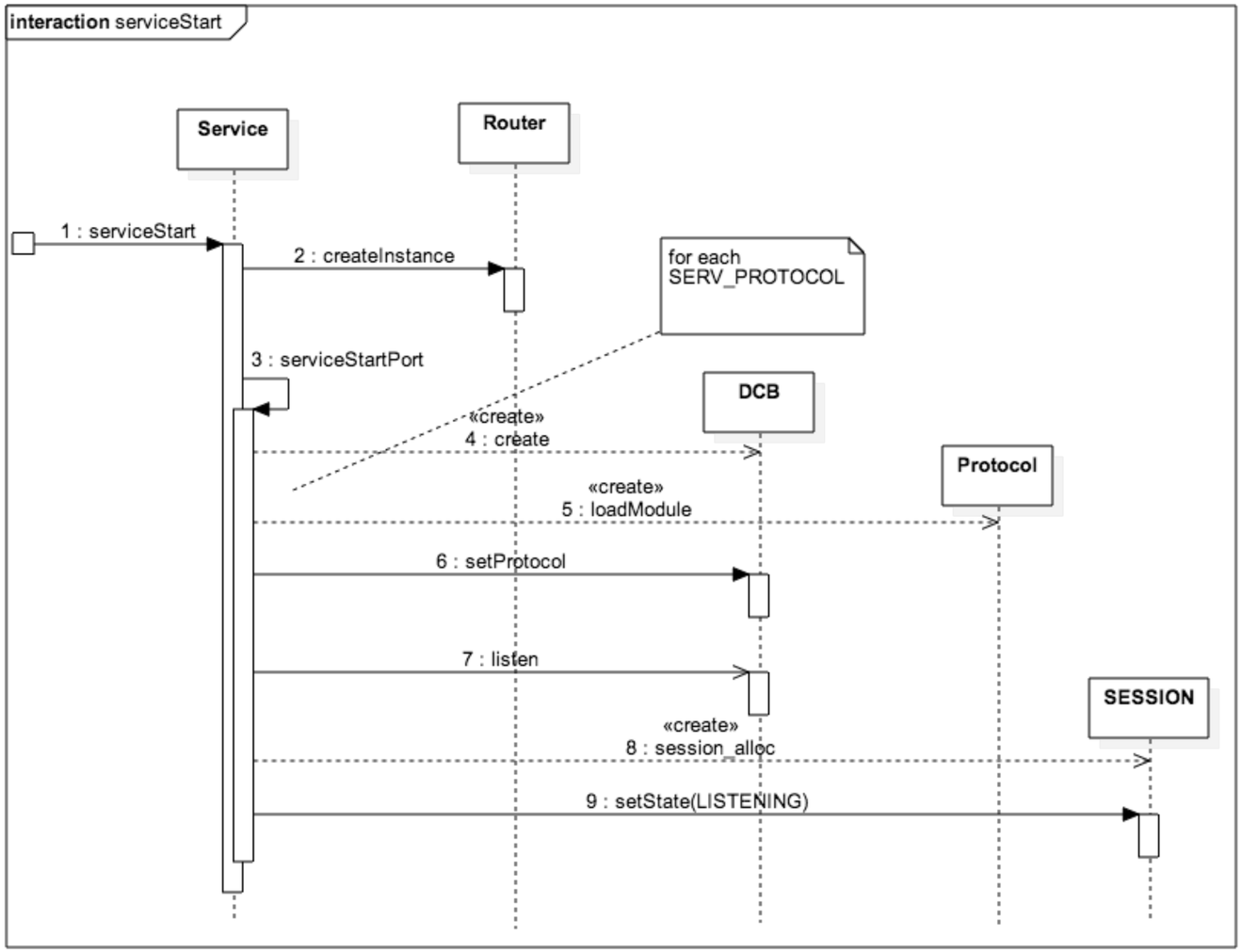
Service Listeners



- The SERV_PROTOCOL structure represents listening ports
- A service may have multiple protocol/port entries
- A special session is created for the listener
- Listener sessions are created for each SERV_PROTOCOL entry

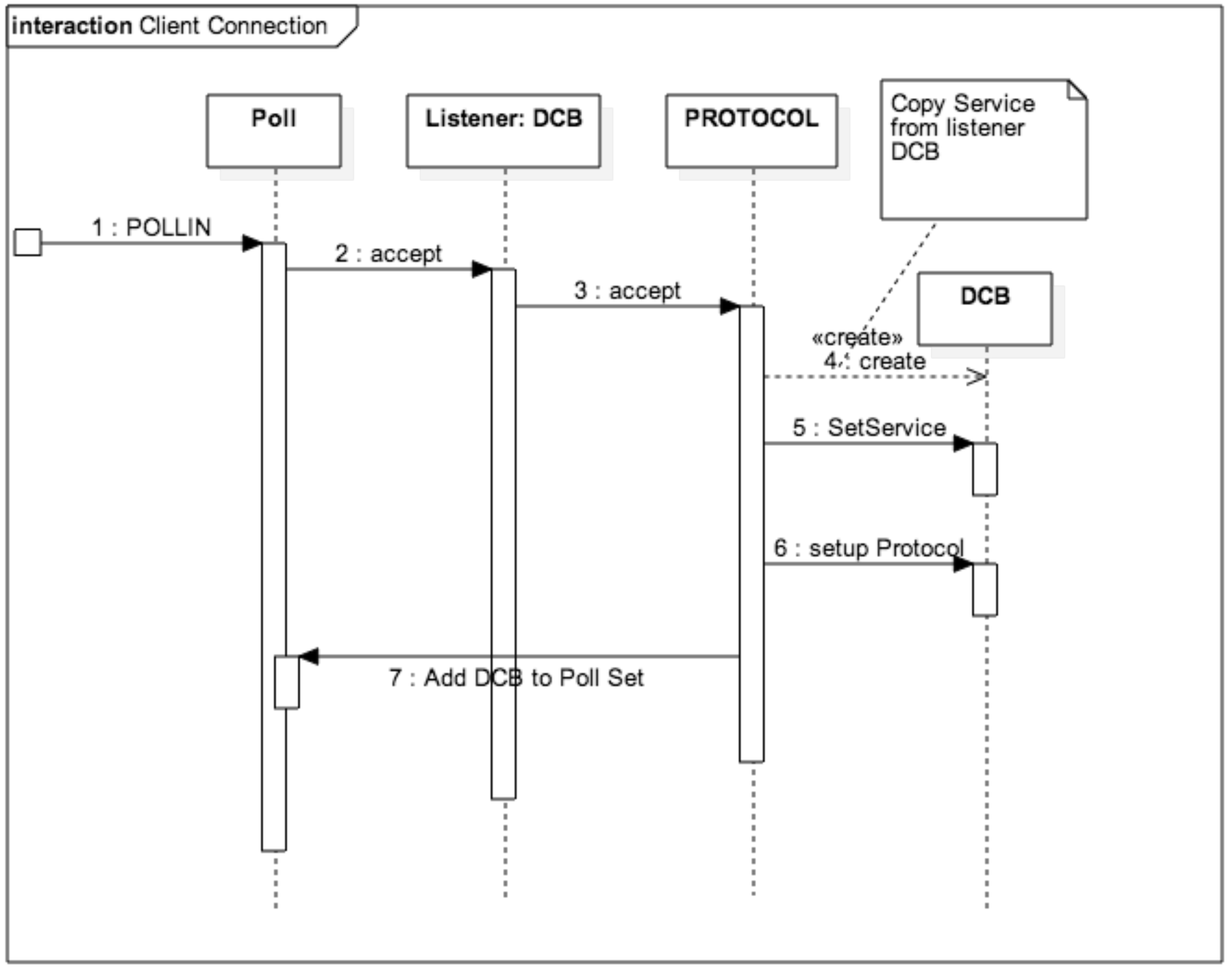
Starting Listeners

- Starting a service
  - Creates an instance of the router that is used by all sessions of this service
  - Loads the protocol modules
  - Creates a DCB for the listener
  - Creates the Listener Session

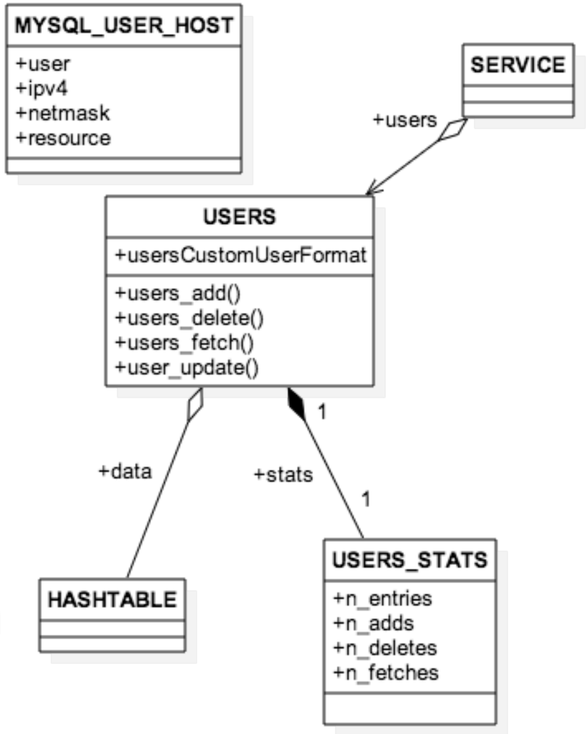


New Connection

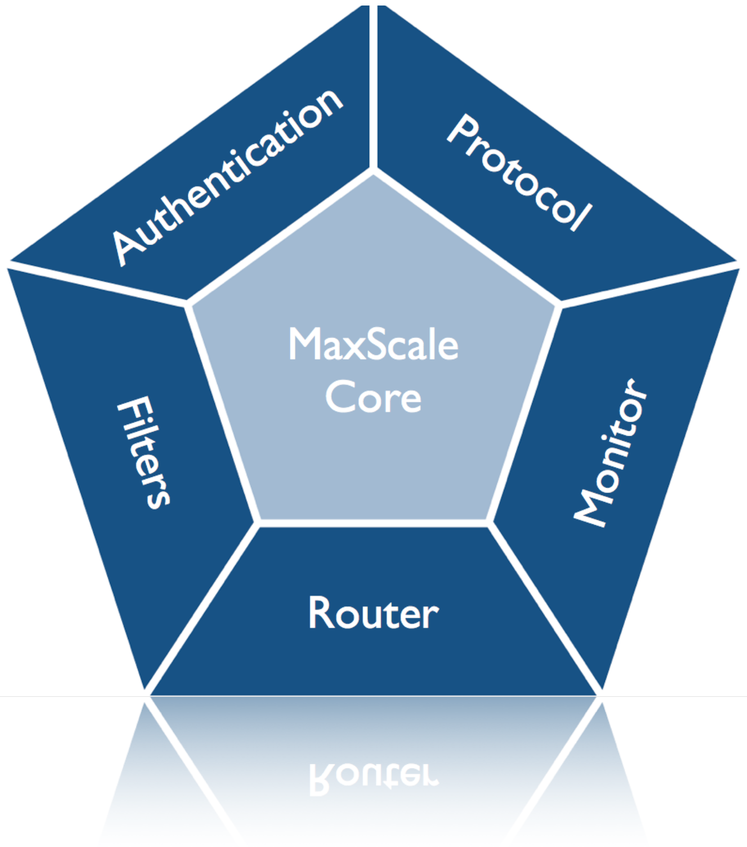
- New Connections cause POLLIN events on the listener DCB
- Protocol does connect and creates DCB
- Service and Protocol copied from listener
- Session not created until authentication is completed



Users



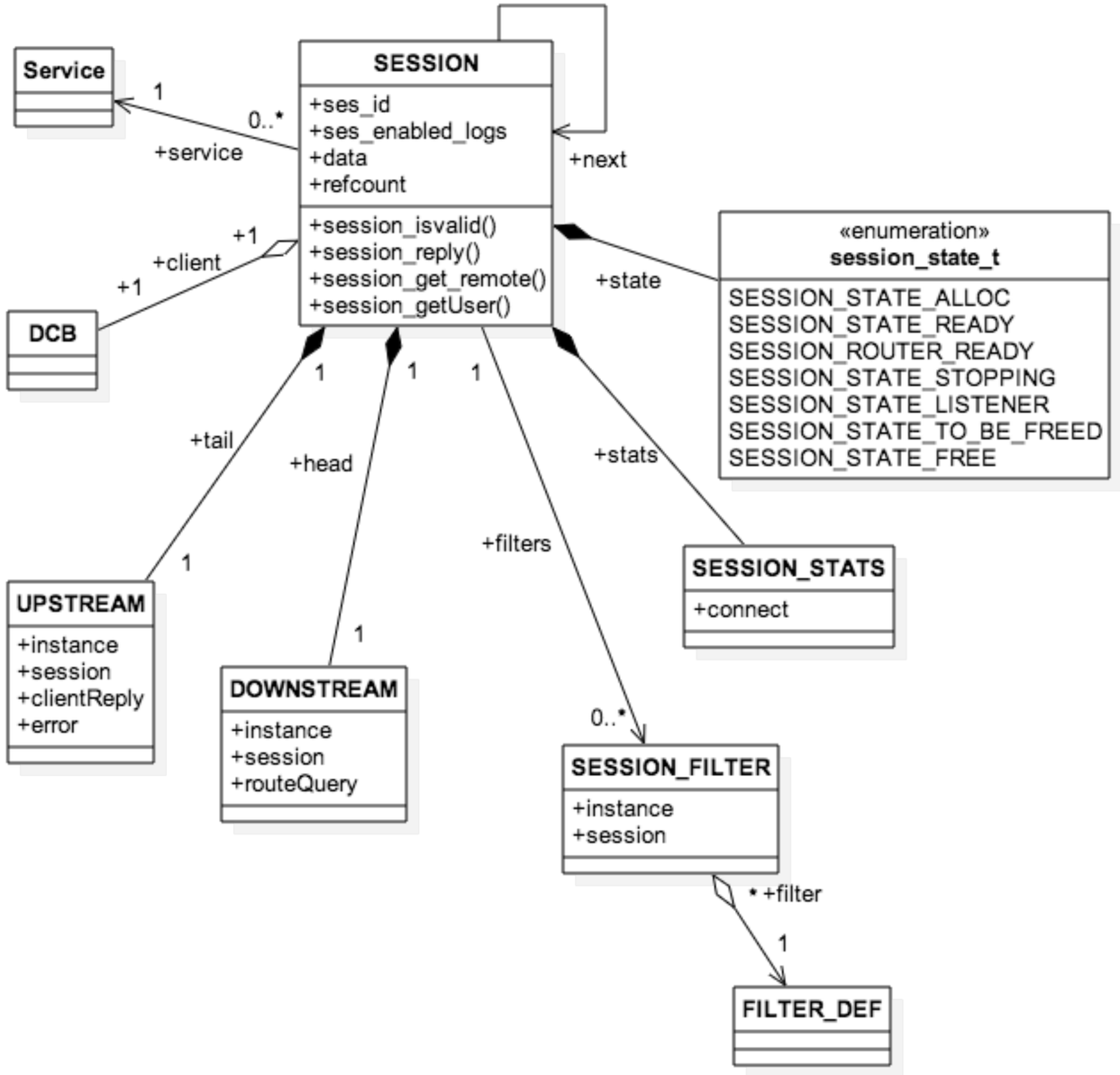
- Users are connected to the service
  - Probably should move to an authentication object when authentication plugins become available
- Each service has a distinct set of users
- Uses HASHTABLE to store user data
- Key may be complex structure, e.g. MYSQL_USER_HOST
- SERVICE_USER credentials used to load users from MySQL
- Not limited to MySQL users



Session View

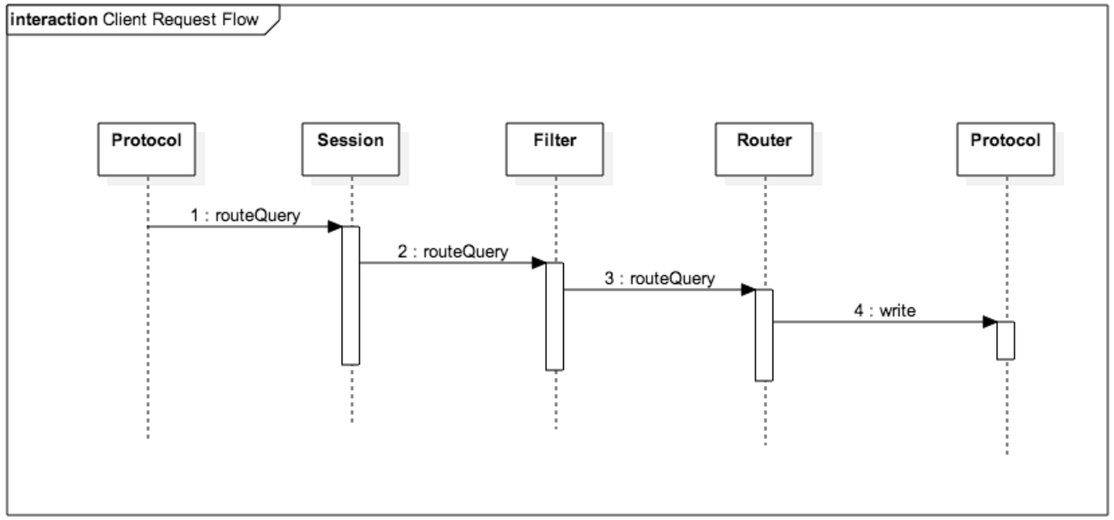
Session View

- Dynamic View
- One session per client connection
  - Except in cases of pseudo clients (e.g. tee filter)
- UPSTREAM & DOWNSTREAM are the session plumbing



Downstream Plumbing

- Downstream - client request to backend
- Each component has pointer to next
- Interface at each level is identical



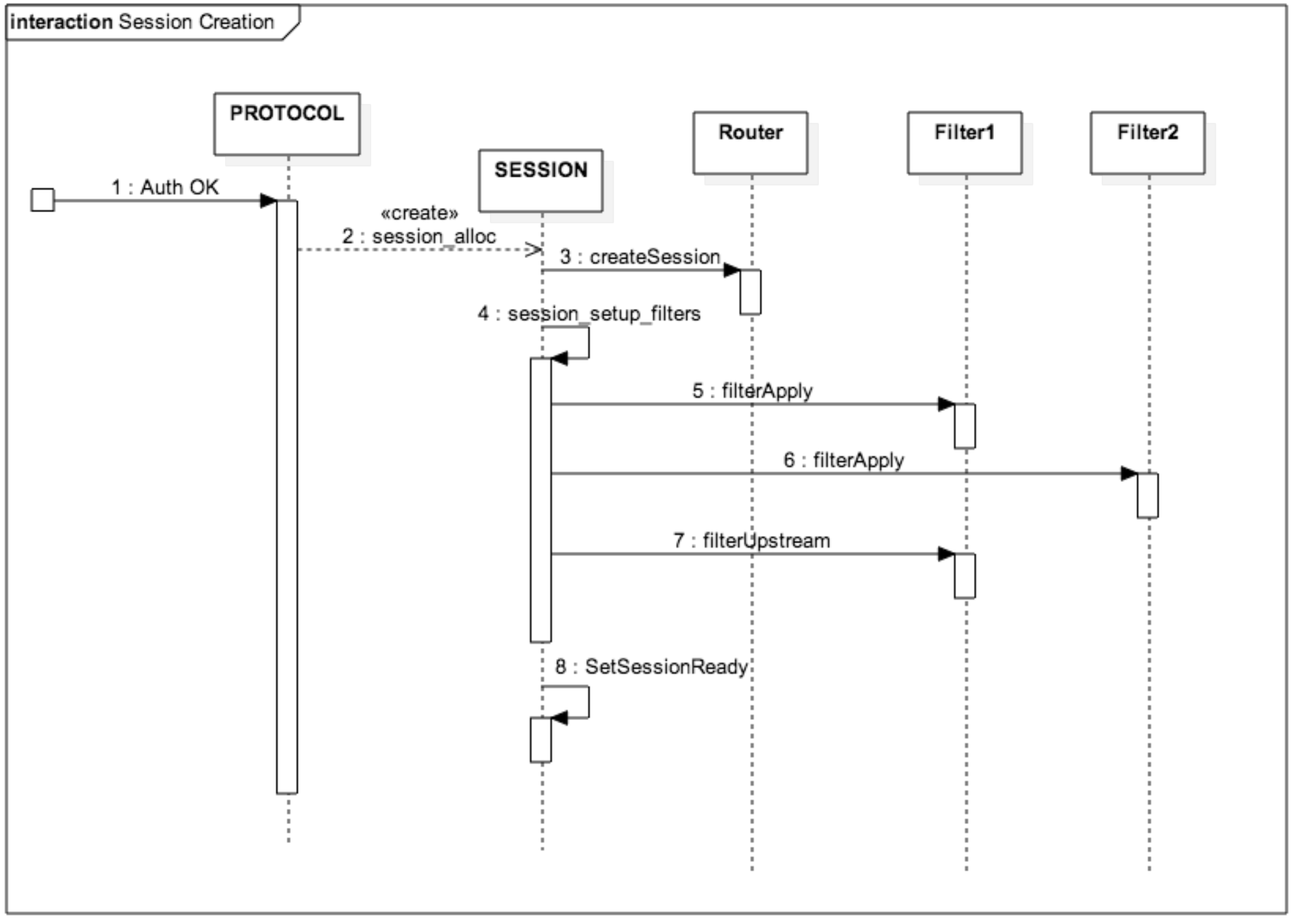
- Each level has opaque instance and session pointers that are passed to the next
- Each layer may modify request content or return without forwarding

Plumbing (Contd.)

- Upstream plumbing is converse of downstream
- Participation in upstream is optional - a filter may not have an upstream interface
- Upstream and downstream structure chain setup when a session is created

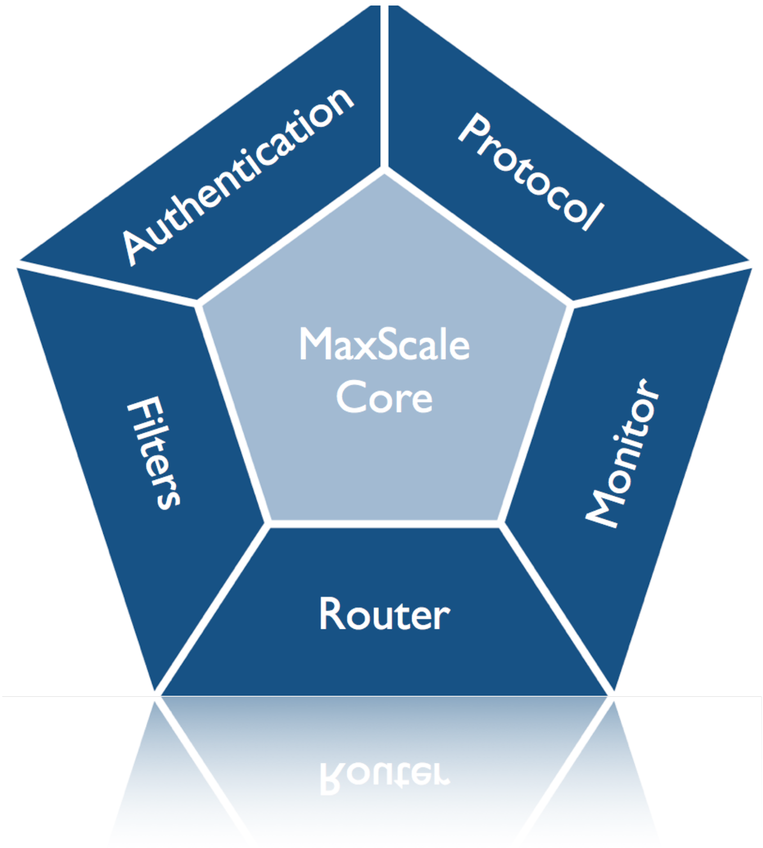
Session Creation

- Triggered by authentication success
- Creates router session
- Sets up plumbing for filters
- Backend connections handled in router



Database Connections

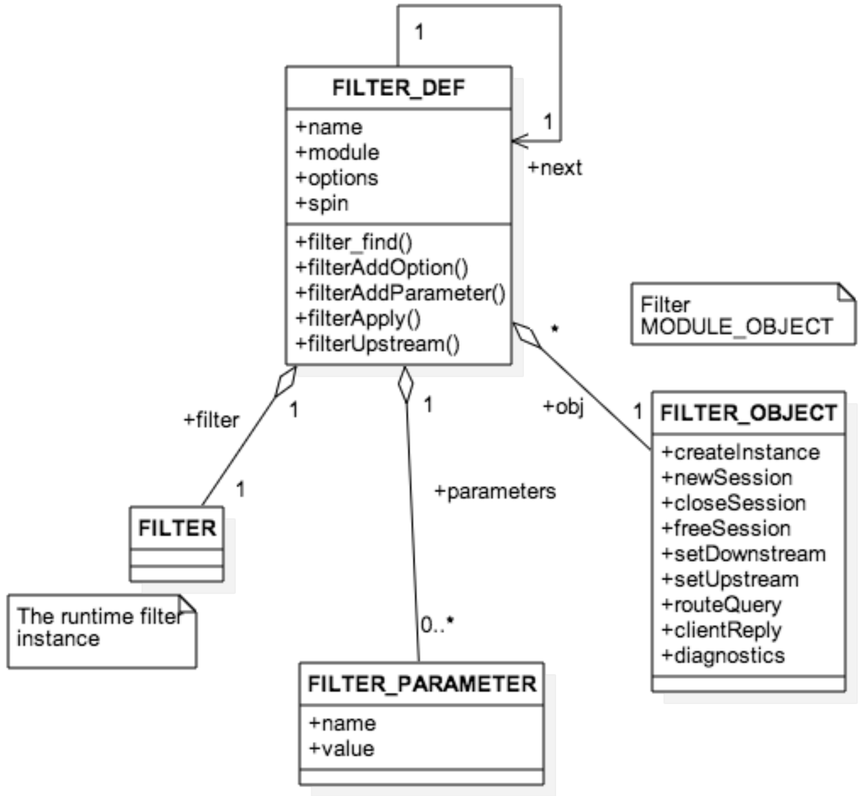
- Database connections **are not** created by the session
- The router manages the backend database connections
  - Only the router knows what the requirements are in the backend
- Database connections can be opened and closed at any point in the session lifetime


Authentication
Protocol
MaxScale
Core
Filters
Monitor
Router

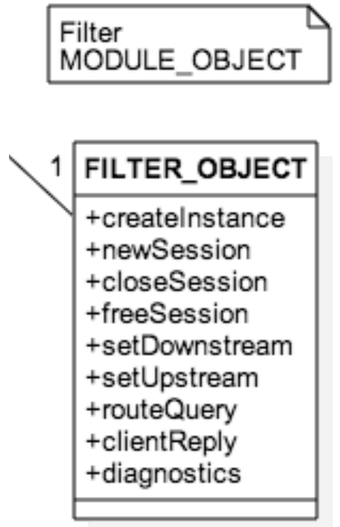
Filters

Filters

- Filters are probably the simplest plugins to write
  - Best advice - start with one of the examples and modify it
- Filters similar to other plugin modules
  - Instances - represent a filter module plus configuration
  - Sessions - A use of a filter in a client session
- Filter module object similar to that of a router
  - Additional plumbing interface
  - Better handling of parameters in configuration

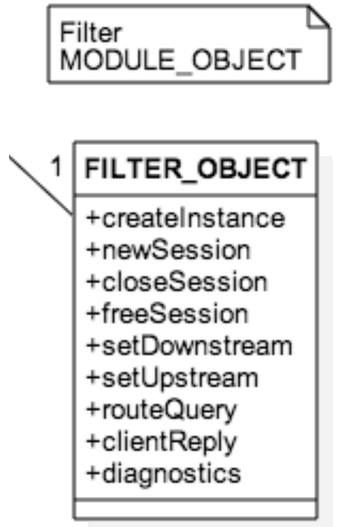


Filters - Create Instance



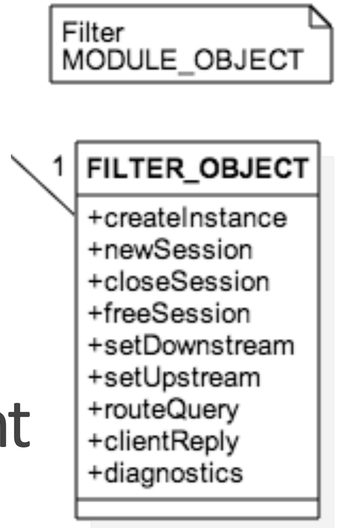
- Create instance is called for each filter defined in configuration
- It tailors the filter to the configuration
- Main task is to create and population the instance structure
- The instance structure contains everything that is common to all sessions using this filter
- Instance structure passed to every other entry point

Filter - New Session



- The newSession entry point is called when a connection is created
- Primary role of newSession is to create and return the session structure
- newSession is called with the instance structure
- The session holds all the state and data related to this particular connection and its use of the filter

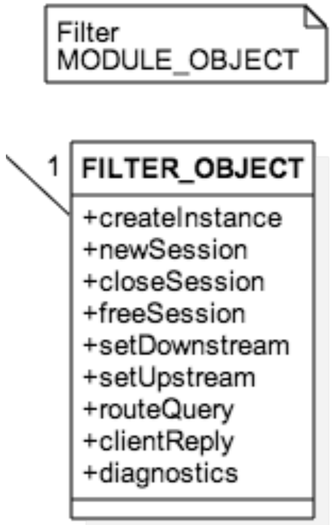
Filter - Set Downstream



- The Filter setDownstream entry point is called to configure next element in the chain
- Passed the instance, session and downstream component information
- Usually stores this in the session structure

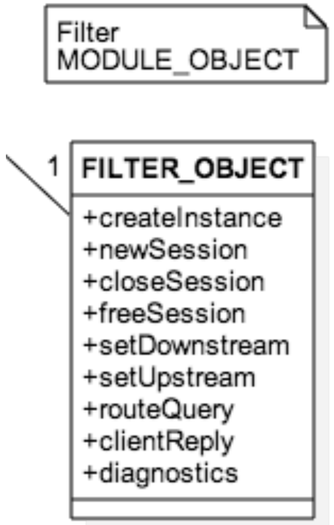
```
/**
 * Set the downstream component for this filter.
 *
 * @param instance      The filter instance data
 * @param session       The session being closed
 * @param downstream    The downstream filter or router
 */
static void
setDownstream(FILTER *instance, void *session, DOWNSTREAM *downstream)
{
MYFILTER_SESSION   *my_session = (MYFILTER_SESSION *)session;

        my_session->down = *downstream;
}
```

Filter - Set Upstream



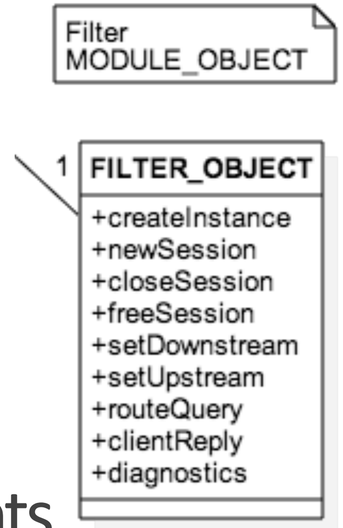
- Optional, only required if results are filtered
- NULL entry point is defined in module object if not required
- Stores the upstream data for returning the result set

Filter - Route Query



- Called for every query packet received
- Always passed instance and session structure
- No guarantees regarding completeness of the request
  - If the filter needs the entire request to be contiguous it must handle this
  - A single packet could contain more than one statement
  - If the filter needs to parse the request it must do so, unless the buffer already has query classifier data attached to it

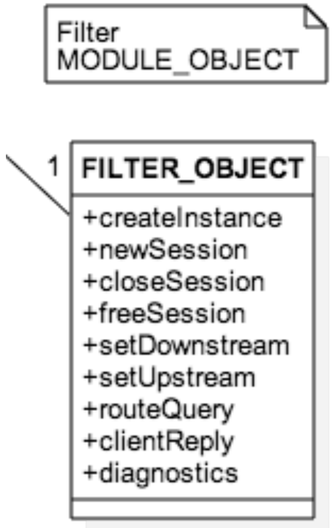
Filter - Route Query (contd.)



- Filters should attach any parse data to the GWBUF for use downstream
- Filters may add hints to be used by the router or downstream components
- Once the filter has manipulated the request it should be passed downstream
- Filters may block downstream processing

```
/**
 * The routeQuery entry point. This is passed the query buffer
 * to which the filter should be applied. Once applied the
 * query should normally be passed to the downstream component
 * (filter or router) in the filter chain.
 *
 * @param instance      The filter instance data
 * @param session       The filter session
 * @param queue         The query data
 */
static  int
routeQuery(FILTER *instance, void *session, GWBUF *queue)
{
MYFILTER_INSTANCE  *my_instance = (MYFILTER_INSTANCE *)instance;
MYFILTER_SESSION   *my_session = (MYFILTER_SESSION *)session;

    …
    …
    return my_session->down.routeQuery(my_session->down.instance,
                        my_session->down.session, queue);
}
```

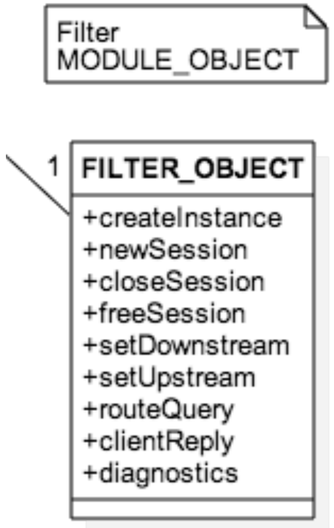
Filter - Client Reply



- Only required if result set filtering is enabled
- Very similar to routeQuery but the buffer is the result set
- On completion results routed upstream

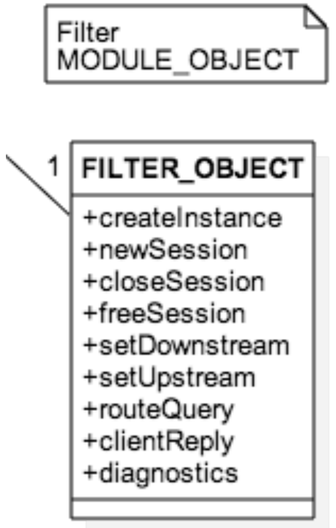
```
/**
 * The clientReply entry point. This is passed the result set buffer
 * to which the filter should be applied. Once applied the
 * query should normally be passed to the upstream component
 * (filter or session) in the filter chain.
 *
 * @param instance      The filter instance data
 * @param session       The filter session
 * @param queue         The result set data
 */
static  int
clientReply(FILTER *instance, void *session, GWBUF *queue)
{
MYFILTER_INSTANCE  *my_instance = (MYFILTER_INSTANCE *)instance;
MYFILTER_SESSION   *my_session = (MYFILTER_SESSION *)session;

    …
    …
    return my_session->up.clientReply(my_session->down.instance,
                          my_session->down.session, queue);
}
```

Filter - Close Session



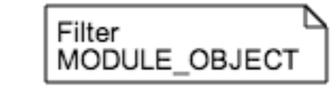
- Close session called after the last request in the session has been sent
- Close is called on every component in the chain separately
- The session may still be accessed for responses and diagnostics after this call

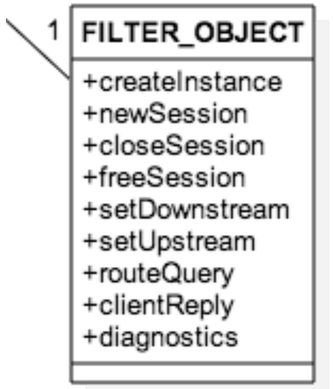
Filter - Free Session



- Last call made for this session
- Should free any session specific data

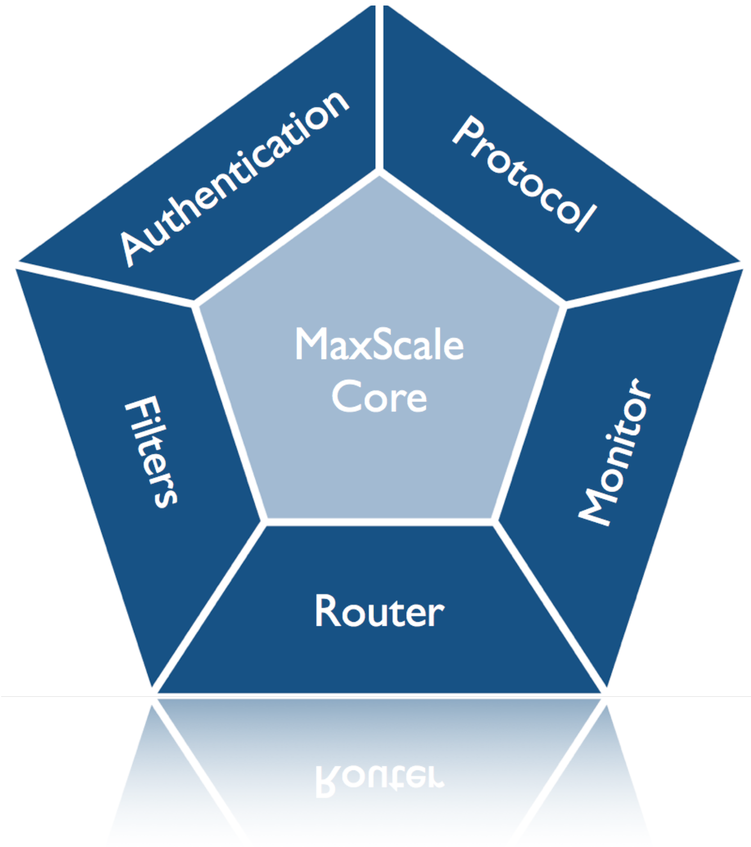
Filter - Diagnostics





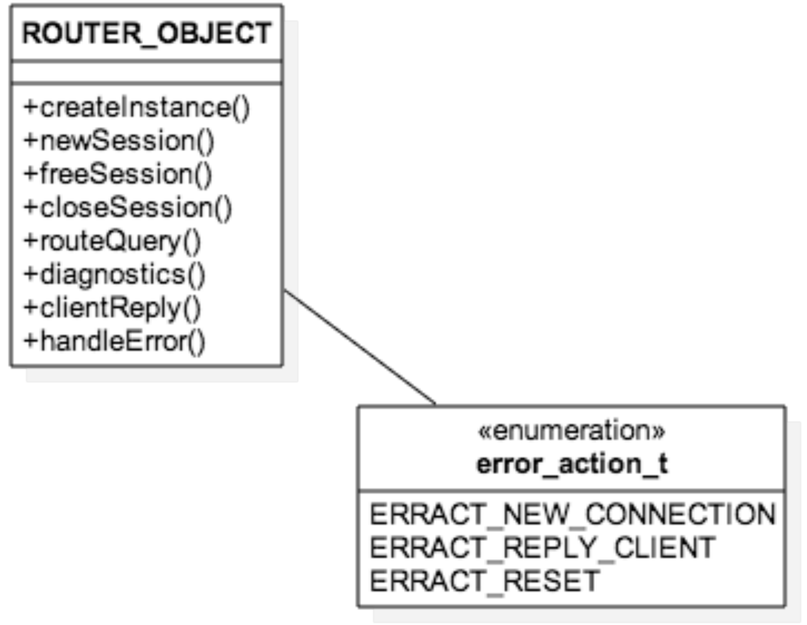
- Called to print diagnostic/monitoring data
- Passed a DCB to print data to
- Called as part of “show service” command to a service or “show session” for a particular session
- If called with a NULL session then print instance diagnostics
- If called with a session then part of “show session” call

```
/**
 * Diagnostics routine
 *
 * If session is NULL then print diagnostics on the filter
 * instance as a whole, otherwise print diagnostics for the
 * particular session.
 *
 * @param       instance        The filter instance
 * @param       session         Filter session, may be NULL
 * @param       dcb             The DCB for diagnostic output
 */
static  void
diagnostic(FILTER *instance, void *session, DCB *dcb)
{
MYFILTER_INSTANCE   *my_instance = (MYFILTER_INSTANCE *)instance;
MYFILTER_SESSION    *my_session = (MYFILTER_SESSION *)session;

        dcb_printf(dcb, …
```


Authentication
Protocol
MaxScale
Core
Filters
Monitor
Router

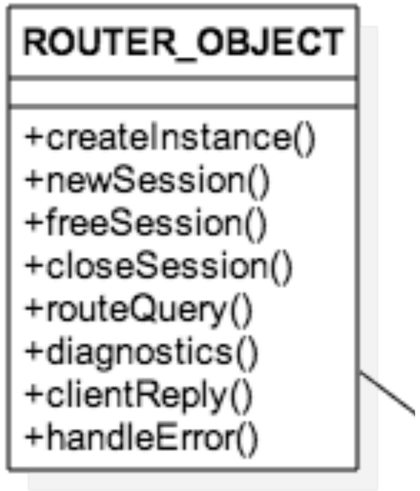
Routers

Router Interface



- Module Object provides familiar entry points
- Routers have instances and sessions
- The instance configures the router to the service definition
- The session manages the data for each individual client connection

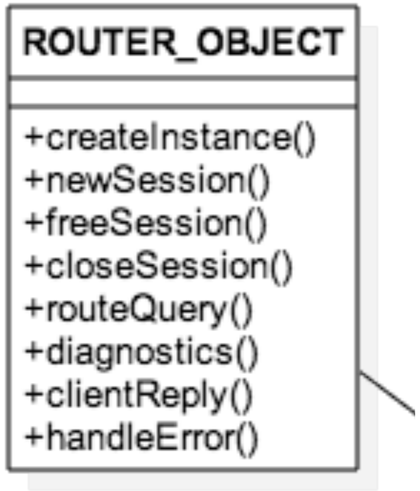
Router - Create Instance



- Creates instance structure for each router
- One instance of a router per service
- Passed router options from service definition
- Returns the instance data passed to other calls
- May do other initialisation

```
/**
 * Create an instance of the router for a particular service
 * within MaxScale.
 *
 * @param service       The service this router is being create for
 * @param options       An array of options for this query router
 *
 * @return The instance data for this new instance
 */
static  ROUTER  *
createInstance(SERVICE *service, char **options)
{
MYROUTER_INSTANCE    *inst;

    if ((inst = (MYROUTER_INSTANCE)calloc(1,
                                sizeof(MYROUTER_INSTANCE))) == NULL)
        return NULL;
    …
    return inst;
}
```

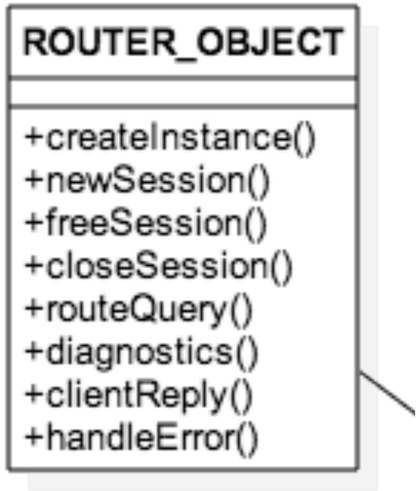
Router - New Session



- Called on client connection
- Creates session data related to that connections use of the router
- Called with instance and session
- Performs initialisation
- Connection based router make routing decision
- Returns the newly created router session

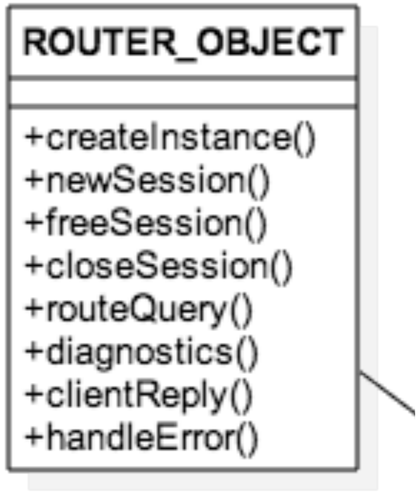
```
/**
 * Associate a new session with this instance of the router.
 *
 *
 * @param instance      The router instance data
 * @param session       The session itself
 * @return              Session specific data for this session
 */
static  void    *
newSession(ROUTER *instance, SESSION *session)
{
ROUTER_INSTANCE         *inst = (ROUTER_INSTANCE *)instance;
ROUTER_SESSION          *my_session;

    if ((my_session = (ROUTER_SESSION *)calloc(1,
                        sizeof(ROUTER_SESSION))) == NULL)
        return NULL;
    …
    return my_session;
}
```

Router - Close Session



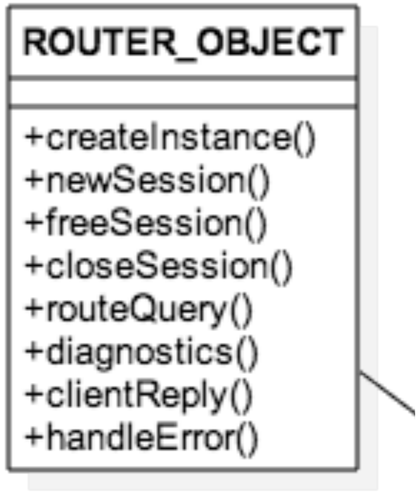
- Called when the last request has been sent
- Signals router to close backend connections
- Responses may still arrive (on other threads) and diagnostics may be called

Router - Free Session



- Final call in lifecycle of a router
- All resources should be deallocated

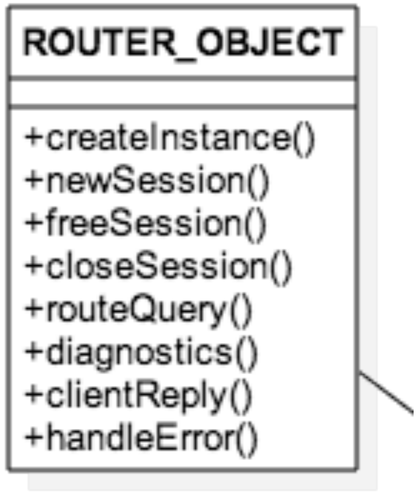
Router - Route Query



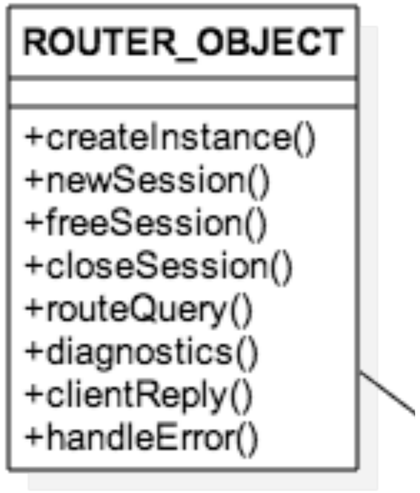
- Called for every packet received
- End of filter chain
- Always passed instance and session structure
- No guarantees regarding completeness of the request
  - If the router needs the entire request to be contiguous it must handle this
  - A single packet could contain more than one statement
  - If the router needs to parse the request it must do so, unless the buffer already has query classifier data attached to it

Router - Route Query (contd.)

- Connection based router merely forwards request to chosen backend
- Statement based routers must make choice of backend
  - Parse query if not already done
  - Examine server states and match with router options
  - Handle backend connection requirements
  - Handle session commands

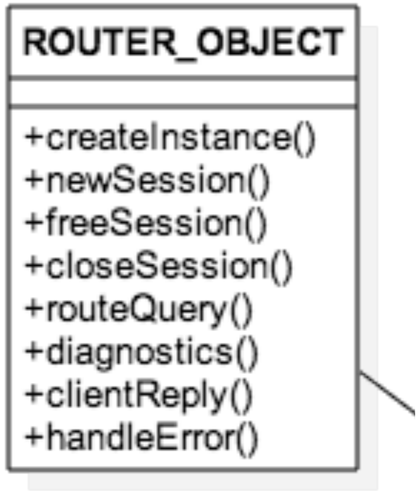


Router - Client Reply



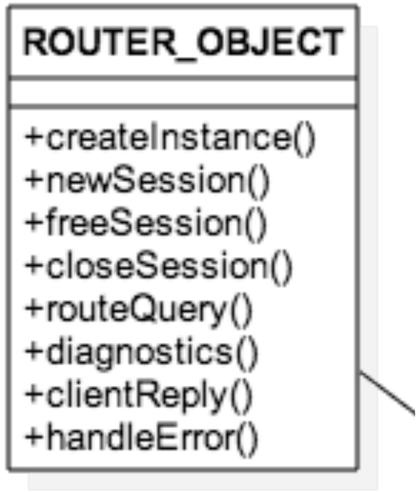
- Handle result sets from backend database
- Filters multiple result sets from session commands
- Pass result upstream via filter chain to session

Router - Handle Error

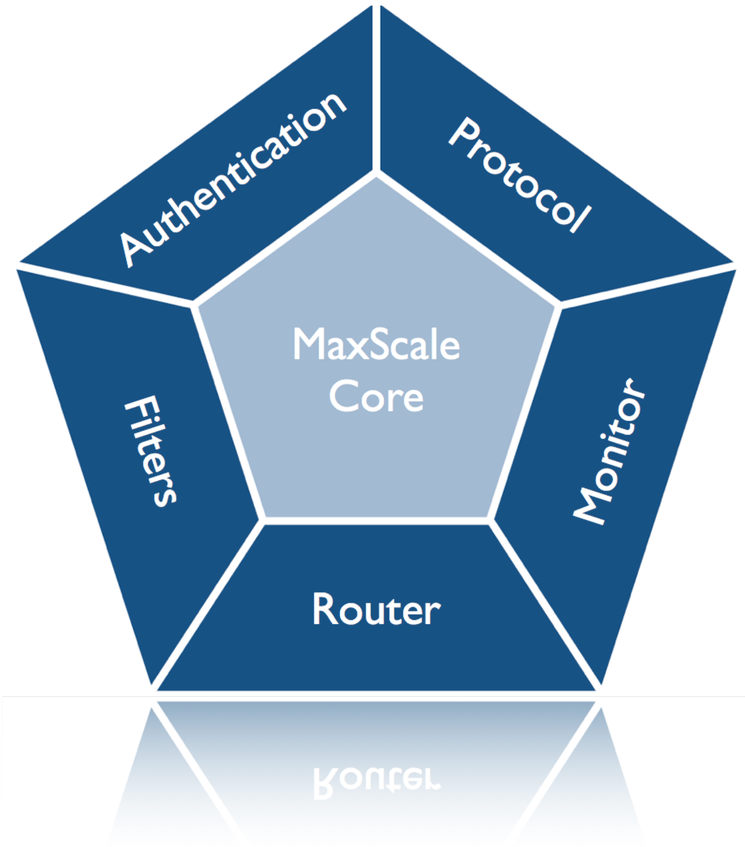


- Handle an error returned from the backend server or protocol layer
- Three options for handling the error
  - May simply pass the error upstream
  - Close connection and shutdown session
  - Open a new connection to either the same of possibly different backend
- May mark error to be handled upstream as well

Router - Diagnostics



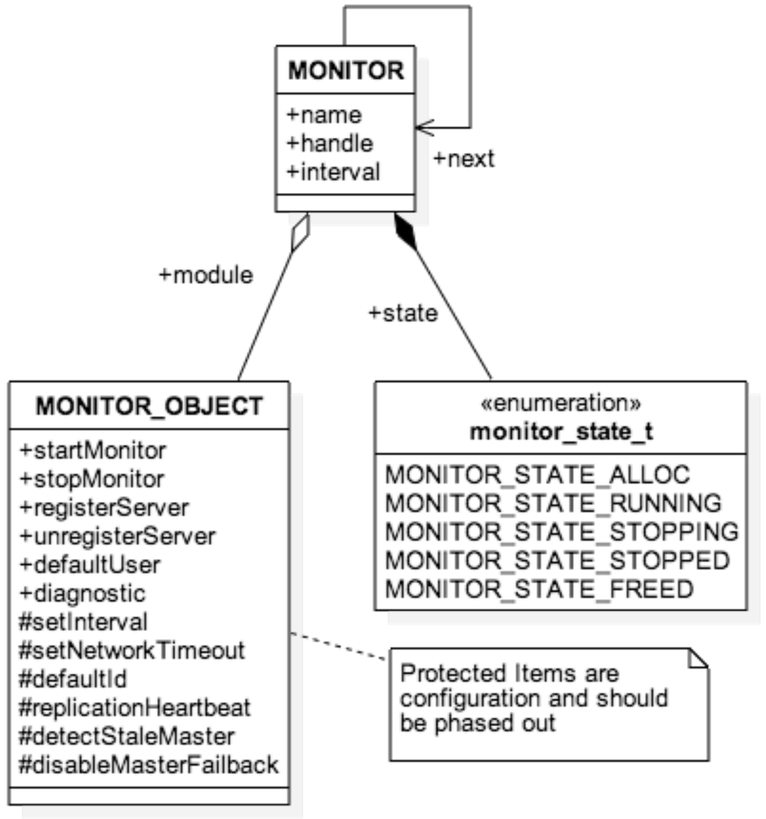
- Only passed an instance, no session
- Print diagnostic and instance data
- Passed a DCB on which to print the data
- Called as part of show service command


Authentication
Protocol
MaxScale
Core
Filters
Monitor
Router

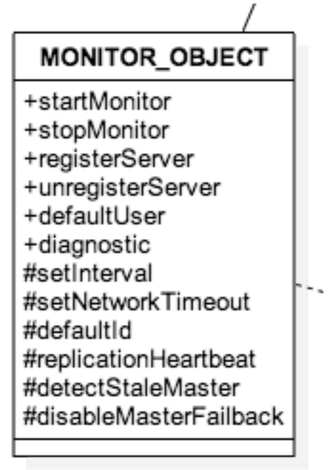
Monitors

Monitors

- Monitors are different to other plugins
- Not related to a service or set of sessions
- Monitors relate to sets of servers
- Monitors run independently of request threads
  - Each monitor runs in its own thread

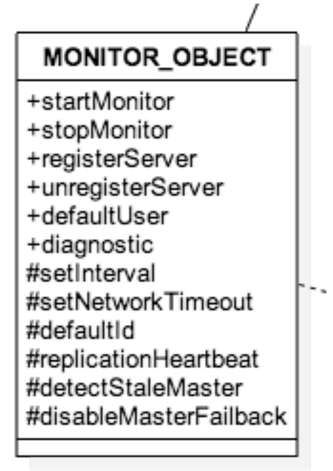


Monitor Module Object



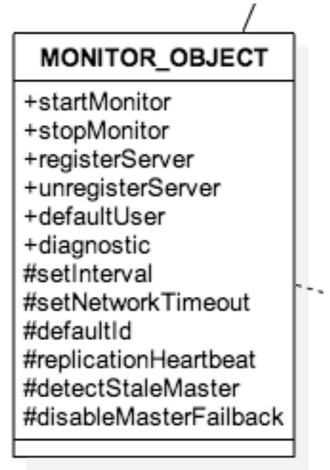
- Some configuration functions have been added to the interface
- Better to remove these
  - Not universally applicable
  - Constrains future monitors

Monitor - Start Monitor



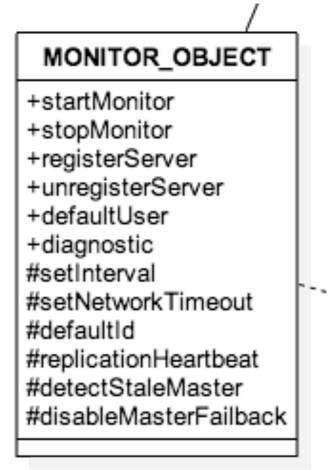
- Creates thread and starts monitoring loop
- Passed a NULL on first invocation
- If restarting a stopped monitor passed handle previously returned
- Returns a handle that is passed to all subsequent calls
  - The handle is essentially the monitor instance structure
- Do not use static data - multiple instance of a monitor may exist in same MaxScale

Monitor - Stop Monitor



- Stop a running monitor
- Does not stop thread
- Stops the actual processing and updating of server state
- Restart by calling startMonitor entry point with previously allocated handle

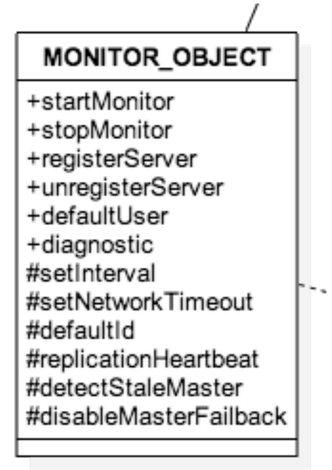
Monitor - Register Server



- Add a server to the list of servers that should be monitored
- Called with the monitor handle and pointer to a server

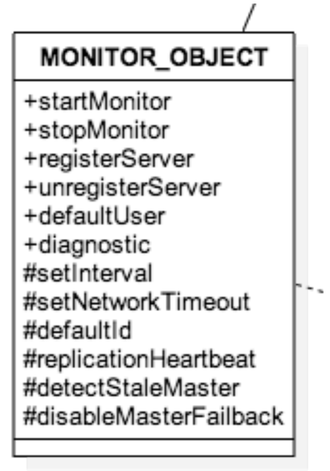
```
/**
 * Register a server that must be added to the monitored servers for
 * a monitoring module.
 *
 * @param arg   A handle on the running monitor module
 * @param server        The server to add
 */
static  void
registerServer(void *arg, SERVER *server)
{
MY_MONITOR   *handle = (MY_MONITOR *)arg;
MONITOR_SERVERS *ptr, *db;

        if ((db = (MONITOR_SERVERS *)malloc(sizeof(MONITOR_SERVERS)))
                                == NULL)
                return;
        db->server = server;
        …
```

Monitor - Unregister Server



- Converse of registerServer
- Stop monitoring the server

Monitor - Diagnostics



- Print monitor diagnostics to a supplied DCB
- Called as show monitor command