OSS Developer's Guide

Michael Schöttner, Marc-Florian Müller, Kim-Thomas Möller
(Universität Düsseldorf)

24. April 2009

1 Introduction

The Object Sharing Service (OSS) implements shared objects for grid applications. OSS is built as a shared library. Linking OSS to user applications allows sharing of objects residing in volatile memory across multiple nodes in the grid. An object in this context is a replicated volatile memory region, dynamically allocated by an application or mapped into memory from a file.

Objects may contain scalars, references, and code. Therefore, OSS handles concurrent read and write access to objects and maintains the consistency of replicated objects. Persistence for objects stored in files are provided by XtreemFS, fault tolerance in contrast is provided by the grid checkpointing mechanisms developed in WP3.3. Currently, OSS supports IA32 and AMD64 compatible processors.

This report is structured as follows. Section **??** describes how OSS implements the XOSAGA API. Section **??** explains OSS' modular architecture and its network protocol. Section **??** documents the internal interfaces of the modules. Finally, section **??** describes step by step how to extend OSS with custom consistency models.

2 API

OSS's services are available via the XOSAGA API [**?**]. Besides, we have implemented a POSIX support library, which emulates POSIX's malloc and free calls for unmodified legacy applications. Both the XOSAGA API and the interface of the POSIX support library are based on the internal OSS interface which has been described in the OSS interface and user guide [**?**].

3 Architecture

Developed as a modularized system, OSS contains three main modules *cache management, consistency models* and *network communication* (see figure **??**).

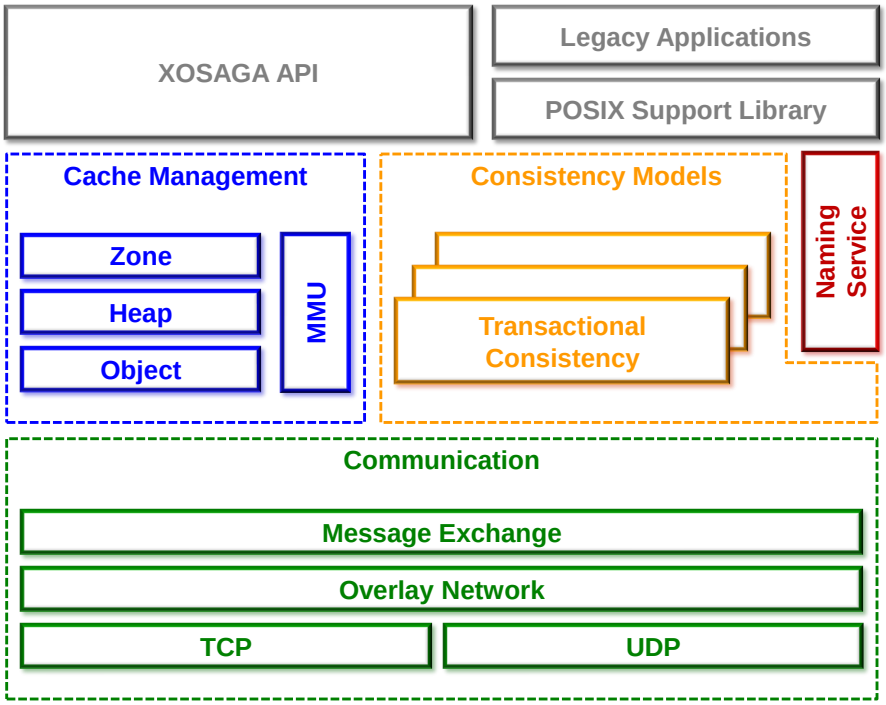

Abbildung 1: OSS architecture

3.1 Cache Management

OSS's cache management allocates objects from a global distributed object space. At a high level of abstraction, objects are represented as chunks of memory. The interface to the cache management is modeled after dynamic memory allocation on the heap, a technique well-known to most programmers. OSS does not interprete the content of objects, such that it allows to share objects in virtually all programming languages. The cache management comprises object allocation, replication and basic synchronization services used by the consistency models.

3.2 Consistency Models

OSS is designed to support multiple consistency models for shared data synchronization. Application developers are able to allocate multiple objects, each coupled

with a consistency model suitable for application semantics. Currently, OSS supports *strong, transactional* and *explicit consistency*.

Strong Consistency Using the *strong consistency* memory modifications are immediately visible on subsequent reads. It is implemented using a modified version of the MESI cache coherence protocol, where the state *Exclusive* has been omitted. Strong consistent objects (in current OSS version allocated at page granularity) can obtain the following states (see figure **??**):

`MODIFIED EXCLUSIVE` Object is modified and exclusive accessible by one node

`SHARED` Object is shared among multiple nodes

`INVALID` Object is invalid

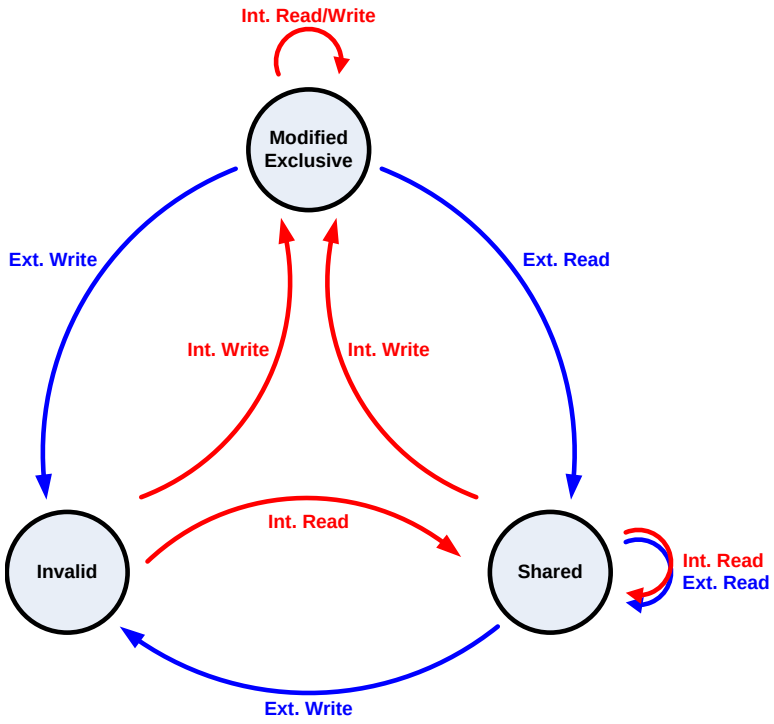

Abbildung 2: Finite state machine of strong consistency

In general, if a node gains write access to a specific object, it changes into modified state. Simultaneously, on all other nodes the object changes into invalid state. Gaining read access instead, the object changes into shared state. Any other node having exclusive access also changes into shared state.

Transactional Consistency *Transactional consistency* also provides strong consistency but multiple operations are bundled into atomic transactions. Possibly occuring conflicts among transactions will be resolved transparently to applications in the background. The developer has only to define transaction boundaries by placing the following two function calls into the program code, defining begin and end of transactions:

```
oss_transaction_id_t oss_bot(...)

int oss_eot(oss_transaction_id_t)
```

Transactional consistent objects can obtain the following states (see figure **??**):

`UNBOUND` Memory page has not been accessed

`BOUND READ` Memory page has been accessed for reading

`BOUND WRITE` Memory page has been accessed for writing

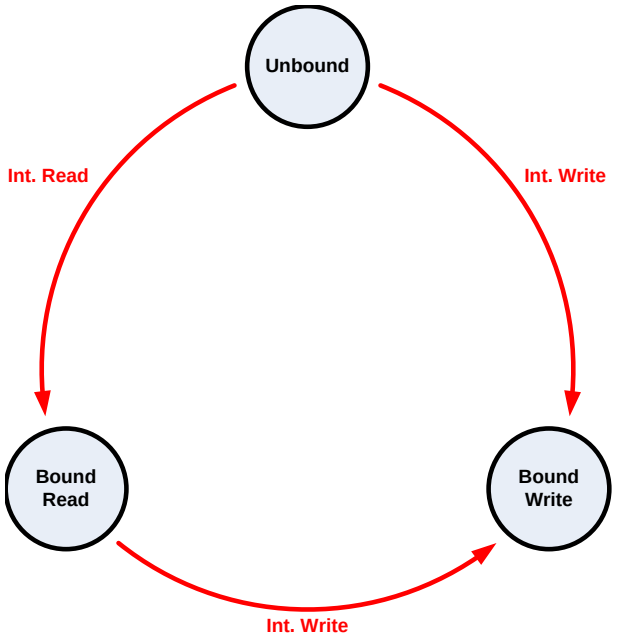

Abbildung 3: Finite state machine of transactional consistency

Explicit Consistency *Explicit consistency* is a pseudo consistency model. Memory allocated under this consistency constraints will never be synchronized and behaves like local allocated memory (e.g. memory allocation via *malloc*). But the memory allocation scheme still follows the semantics of OSS. As in other consistency models, objects using this consistency model reside at the same memory address on all peers.

3.3 Name Service

OSS contains a simple internal name service, which applications can use to store and retrieve object IDs. The name service has a tree structure, with slashes (/) separating directory levels. An application or OSS module can set a value for a name by calling `oss_nameservice_set` and retrieve a value by calling `oss_nameservice_get`. A value that has not yet been set is treated as object ID `NULL`.

3.4 Network Communication

The network module has two layers. The lower layer implements the binding of transport protocols like TCP (Transmission Control Protocol) or UDP (User Datagram Protocol) to the *overlay network* and assembles incoming data fragments to PDUs (Protocol Data Units). Furthermore, it implements fault tolerance mechanisms if not supported by the transport protocol itself. Currently, OSS uses TCP, only.

The upper layer implements functionality for establishing and managing the overlay network. Peers will be grouped together, coordinated by one *super peer* node which manages inter group communication and group internal tasks (e.g. transaction validation). The super peer will be elected on the basis of its properties (e.g. performance, network latency and bandwidth, average cpu load, ...). Moreover the overlay network routes messages among nodes and is dynamically reconfigurable by using statistical data collections. The communication module implements an interface to abstract messaging from the underlying network structure.

For efficiency reasons, OSS implements its own binary request/reply network protocol. Every PDU begins with a header followed by an optional payload part. All fields of the PDU are described in detail (see figure **??**)

Type Primary type of a network message. This field coincide with the modules which have registered a specific message type. All messages of a type are passed to the registered handler of a module. The communication module distinguishes request and response messages by inspecting the MSB (Most

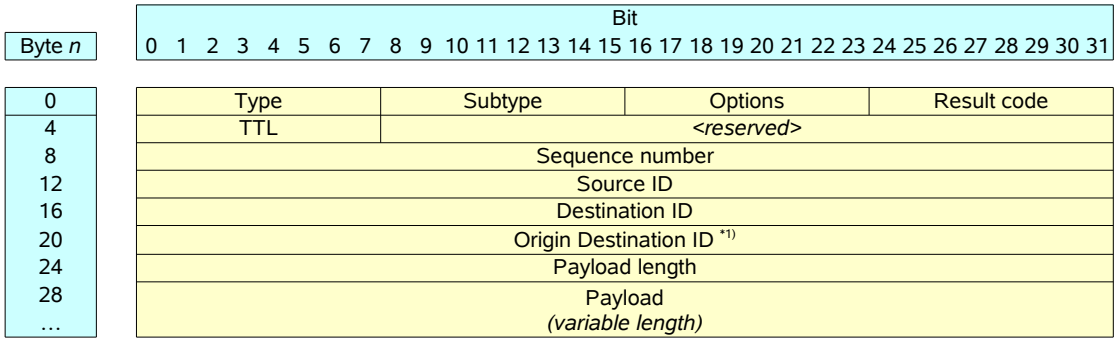

Abbildung 4: Structure of Protocol Data Units in OSS

Significant Bit). An unset bit defines a request message, a set bit a response message. A pair of request and response messages differ only in the MSB referred to the type field.

Subtype Subtype of a network message. This field is used by modules to distinguish various messages of the same type.

Options Internally used bit field.

Result code In response messages this field carries the result code of the previously processed network request.

TTL Time-to-Live field, which prevents an endless routing/forwarding of messages in the overlay network. This field is decremented on every message routing/forwarding.

Sequence number Consecutive number for assigning reponse to request messages and to preserve message ordering.

Source ID Node ID of the sender.

Destination ID Node ID of the recipient (modified in case of message forwarding).

Origin dst. ID Node ID of the origin recipient (still unmodified in case of message forwarding).

Payload length Length of payload.

Payload Payload.

4 Internal Interfaces

As mentioned in section **??**, OSS' services are available through the XOSAGA API. The internal interfaces of OSS' modules are used only inside the shared library and therefore will not be exported to user applications. The following sections describe the module interfaces of *cache management* and *network communication*.

4.1 Cache Management

Each object has a unique identifier within the shared object space. To enable efficient parallel allocation of objects, nodes pre-reserve heaps of objects using the distributed allocator. The cache management comprises different allocators to partition the per-node heaps. When creating an object, an application can specify which allocator to use, depending on the object's intended usage. Access control and storage management provide basic mechanisms to keep objects consistent.

4.1.1 High-level object management (object)

The high-level object management module dispatches object creation and object deletion to the respective heap. The module's interface consists of the following four functions:

- The function `memory_alloc` reserves memory for the object on a heap, binds it to the specified consistency model and attributes, and returns a reference to the newly created object.
- The function `memory_free` frees the memory pointed to by the reference.
- The function `memory_mmap` creates an object as a copy-on-write mapping of the specified file. If the file is unspecified, the function creates an anonymous mapping.
- The function `memory_munmap` destroys an object which is a file mapping or an anonymous mapping.

4.1.2 Object allocation (heaps)

During object creation, the high-level object management allows an application to choose the heap on which the object will be allocated. OSS currently implements three heap allocators:

- The *page allocator* always reserves at least one hardware page (4 KB). This allocator perfectly avoids false sharing situations, but it incurs a high memory overhead for small objects.

- The *mspaces allocator* integrates the standard allocator from GNU/Linux's standard C runtime library. This allocator is very efficient in terms of memory usage, but depending on the application's object access pattern, false sharing situations can degrade performance.
- The *millipage allocator* implements the Multiview/Millipage approach avoiding false sharing [**?**]. This allocator tries to avoid false sharing despite low memory overhead. It uses special features from the access control module to efficiently place objects.

4.1.3 Access control and storage management (mmu)

The access control and storage management module abstracts from hardware and operating system features. By means of this module, consistency models access object data and keep track of object state. The following functions are related to information about objects:

- The function `mmu_get_consistency_model` retrieves the consistency model that a given address is bound to.
- The function `mmu_is_valid_address` determines whether an address is known to the local node.
- The functions `mmu_set_state` and `mmu_get_state` access an object's state as defined by the respective consistency model.
- Consistency models can store more information about an object's state using the functions `mmu_set_data` and `mmu_get_data`.
- The functions `mmu_lock`, `mmu_unlock` and `mmu_trylock` synchronize access to object metadata.

The following functions manage the physical backing store for several objects:

- The function `mmu_alloc` is called by the heap allocators to set up the physical backing store for several objects.
- The function `mmu_setup_region` sets up a page-aligned memory region at a specified address. This function applies to locally created regions as well as to remote regions.
- The function `mmu_discover_region` discovers a region using the grid memory allocator and sets it up locally.
- The function `mmu_free` frees a region from physical backing store.

- The function `mmu_foreach_page` runs a function for each page in a memory region.

The storage management functions read or write the content of an object:

- The function `mmu_copy_to_shadow` creates a backup copy for an object, whereas the function `mmu_restore_from_shadow` restores an object from a backup copy. The function `mmu_forget_shadow` discards any backup copy for an object.

- Using the functions `mmu_copyin`, `mmu_copyout` and `mmu_copyout_prefer_shadow`, a consistency model can atomically read or write an object's content.

Using the access control functions, a consistency model can request notification of read or write operations on objects:

- The function `mmu_trap_read_write` configures access control such that the consistency model is notified of reads and writes.

- The function `mmu_trap_write` configures access control such that the consistency model is notified of writes.

- The function `mmu_trap_none` configures access control such that the consistency model does not receive notifications.

4.1.4 Zone allocation (zone)

The zone allocator coordinates coarse-grained reservations among the nodes.

- The functions `allocator_set_root_memory` and `allocator_get_root_memory` access the root memory, which is the basis of the distributed name service within OSS.

- The function `allocator_alloc` reserves a memory region in the global allocator for a node.

- The function `allocator_free` marks a memory region as unused.

- The functions `allocator_discover`, `allocator_get_size`, `allocator_get_consistency_model` `allocator_get_allocator` and `allocator_get_owner` retrieve information about memory regions from the global allocator.

- The function `allocator_is_valid_address` checks whether an address is within the shared object space.

4.2 Network Communication

OSS exchanges network messages among nodes to establish and reorganize the overlay network, synchronize cached objects in memory with respect to the applied consistency model and configure nodes. The interface supports sending, forwarding and droping messages. The send functions are categorized into request and response functions, and in blocking and non blocking functions. When a send function is called, OSS builds a PDU (see figure **??**) which is passed to the overlay network.

4.2.1 Request/Response messaging

In general a conversation among nodes follows a request/reply scheme in which a node sends a request to another node and awaits a response message. In case a request handler itself needs to request further data from other nodes to respond the actual request, the network module supports linking of multiple requests. In particular, if a node receives a request, it is allowed to start a further request from its request handler. The full processing of the request may be deferred until the node has processed the response of the second request. The following functions allow a request/reply comunication among nodes:

- `comm_send_sync_req_to` sends a request to nodes and blocks until the node has processed all response messages.
- `comm_send_async_req_to` is the same as `comm_send_sync_req_to` but never blocks. Response messages are processed in the background.
- `comm_send_async_linked_req_to` sends a new unblocking request from a network request handler. The actual request is linked to the new request and will be reprocessed after the node has processed the response message of the new request. A request handler which calls this function must return with `-E_PDU_LINKED`.
- `comm_send_sync_resp` sends a message in response to a request message. This function will never block.
- `comm_send_async_resp` In the face of response functions will never block, this is only an alias for `comm_send_sync_resp` to keep synchronious/asynchronious messaging semantics.

4.2.2 Informational Messaging

If a message exchange does not await a response, OSS supports sending one way or informational messages.

- `comm_send_async_msg_to` sends a message to nodes. This function will never block and does not await any response messages.

4.2.3 Messaging forwarding

OSS allows forwarding of request messages directly from the request handler of the message itself. It is allowed to forward the same message multiple times. Response messages will be sent directly to the requester without the indirection of the forwarding nodes.

- `comm_forward_req_to` forwards a request to another node.

5 How to implement Consistency Models

This section describes in a few steps how to implement an own consistency model in OSS. Consistency models reside in the folder `src/consistency`, new implementations should also be stored there. First of all, the developer should create a new code and header file for his implementation.

For his own implementation the developer can use the implementation of the *null consistency*[1] as a template. The header file starts with an inclusion guard macro definition which should follow the naming semantics of the already implemented consistency models. At least one declaration (the pointer table of the consistency model itself) must be placed in the header file by adding the following line, where `<NAME>` is a placeholder for the name identifying the consistency model.

5.1 Function Pointer Table

Every consistency model must define a function pointer table for its callback functions:

```
consistency_model_t <NAME>_consistency =
{
.name = "<NAME> consistency",  // Name
.id = oss_<NAME>_consistency,  // Internal identifier
.init = *ptr,                  // Ptr to init function or NULL
.fini = *ptr,                  // Ptr to fini function or NULL
.read_handler = *ptr,          // Ptr to read fault handler
.write_handler = *ptr,         // Ptr to write fault handler
.event_handler = NULL,         // Unused (shall be NULL)
.alloc_handler = *ptr,         // Ptr to memory alloc handler
```

[1] File: *consistency/nc.c* (implementation) and *consistency/nc.h* (declarations)

```
.free_handler = *ptr,          // Ptr to memroy free handler
.mspace = NULL                 // Internally used by mspaces allocator
};
```

5.2 Consistency Model Registration in OSS

To make the consistency model available to user applications, it has to be registered into OSS. This is done by performing the following steps

1. Register the consistency model in the consistency control unit[2]. This is done by appending the pointer of the consistency model's local function pointer table to the global function pointer table in the consistency control unit. Beware of reordering the table entries. Additional consistency models may only be appended to this table.

2. Add an appropriate named entry in the consistency model enumeration in OSS' global header file[3]. The entries in the enumeration must be in the same order as in the global function pointer table. The entry `oss_max_consistency` must always be the last entry in the enumeration.

5.3 Initializer and Finalizer

Code for preinitialization and finalization of the consistency model is placed in initializer and finalizer functions, called before the application starts and after the application terminates. These functions will be called automatically if their function pointers are added to the function pointer table.

```
static void <NAME>_init() {...}

static void <NAME>_fini() {...}
```

5.4 Register PDU handlers

To participate in network communication, the consistency model must register callback functions to handle network request and response messages of a specific message type. It is recommended to register two different callback handlers. The developer has to ensure to register only handlers for message types which have not been registered before, because reregistering a message type will overwrite any previous handler registration. The following steps describe the handler registration for a new message type:

[2]File: *consistency/consctl.c*
[3]File: *oss.h*

1. Add a new request and an appropriate response message type to the PDU header file[4].

   ```
   #define TYPE_<NAME>_REQUEST     ...
   #define TYPE_<NAME>_RESPONSE    (TYPE_<NAME>_REQUEST ...
   ```

2. Include the network communication header file into the implementation file of the new consistency model

   ```
   #include "net/comm.h"
   ```

3. Register the PDU handlers by calling the following function. It is recommended to perform the registration in the consistency model's initialization function.

   ```
   pdu_register_handler(*req_handler, TYPE_<NAME>_REQUEST);
   pdu_register_handler(*resp_handler, TYPE_<NAME>_RESPONSE);
   ```

   Registered handlers have the following signature like the example below:

   ```
   static int req_handler(in_pdu_t *pdu)
   ```

5.5 Writing PDU handlers

The registered request and response handlers must interprete the subtype code of the messages and delegate them to the correct subhandler. For a clean code structure it is recommmended to implement only the interpretation of subtypes in this function and delegate the messages to subhandlers. A sample implementation could look like the following code snippet:

```
switch (pdu->header.subtype) {
  case SUBTYPE_<NAME>:
    return func1(pdu);
    break;
  case ...

  default:   //ignore messages with unknown subtypes
    dbg_printf(0, "invalid pdu subtype\n");
    return PDU_SUCCESS;
}
```

Subhandlers have the same signature like registered request and response handlers.

[4]File: *net/pdu.h*

5.6 PDU subsystem

It is allowed to send new network messages from network handlers. But messaging in this context is covered by the following restrictions

- no use of blocking message functions
- no request messaging from response handlers

Furthermore, network handlers must return immediately after message processing, therefore it is not allowed to block within handlers. The return code of a handler controls the postprocessing of messages. OSS supports the following return codes:

`PDU_SUCCESS` Handler processed succesfully. Request PDUs will be removed from the queue

`-E_PDU_LINKED` Request has been linked with a new request and will be deferred for later reprocessing (request PDUs only)

`-E_PDU_DEFERRED` Request has been deferred for later reprocessing (request PDUs only)[5]

5.7 Page Fault Handler

The page fault handlers implement the consistency model as a finite state machine on a per page basis. OSS signals access violations disjoined regarding read and write faults. The finite state machine consists of multiple page states previously defined by the developer. In conjunction with access right controlling[6] and read/write page faults, raised on access violation, the machine performs its state transitions. The read and write handlers must be added to the local function pointer table and have the following structure:

```
static void <NAME>_read_handler(void *addr,
    struct ucontext *context)
{
  int state = mmu_get_state(addr);

  switch (state) {
    case STATE_1:
      ...
```

[5]This return code may be removed soon and therefore shall not be used for new handlers.

[6]Access rights are configurable for all disjunct virtual memory pages and can allow/disallow read and write access to it.

```
      break;
    case ...

    default:
      dbg_printf(0, "address %p: unknown state detected (%u)\n",
          addr, state);
      exit(EXIT_FAILURE);
  }
}
```

The developer can perform state transitions by controlling the state of the faulted pages with the following two functions:

```
int mmu_get_state(void *addr);

void mmu_set_state(void *addr, int state);
```

Additionally, the access rights of memory pages can be modified via

```
void mmu_trap_none(void *addr);

void mmu_trap_write(void *addr);

void mmu_trap_read_write(void *addr);
```

5.8 Exporting functions to the API

Functions are exported to the Application Programming Interface by writing wrapper functions prefixed with `oss_`. The function declaration must be included into the global header file of OSS, included by the applications. The wrapper code must be included in the corresponding source code file[7].

[7] File: *oss.h* (declaration) and *oss.c* (implementation)