Domain Services Specification

Revision 0.6

February 6, 2006

Please send comments & queries to:

ryan.maeda@sun.com or

eric.sharakan@sun.com

Table of Contents

1 Introduction

1.1 Background

In a Logical Domain environment the ability to discover whether a guest operating system has various capabilities, and be able to remotely direct it to perform various operations is important. Similarly it is equally important for a guest operating system to be able to discover and communicate with its various support services.

Specifically, each guest domain can offer a number of capabilities to its service entity, and similarly the service entity can offer a set of capabilities for use by the guest domain.

Capabilities may include things such as the ability to perform dynamic reconfiguration, or be directed to perform a graceful shutdown or reboot by a service entity.

As a domain transitions through various operational phases, (for example while booting) its capabilities may change. The capabilities of a simple guest OS like OpenBoot are not the same as those of a full blown operating system such as Linux or Solaris. Similarly services that are offered to a domain by its service entity/entities may come and go if, for example, a service processor re-boot occurs.

Consequently it is a requirement that the mechanism for capability discovery and communication must be able to cope with the dynamic nature of both a guest domain and its service entities.

1.2 Domain services protocol

This document describes the protocol by which a guest OS may register its capabilities with its service entity/entities, and vice-versa. The registration process includes independent version negotiation between client and service for each capability

Once a capability has been registered, the domain services protocol then provides a data transport for client and service to communicate directly with each other independently of other capability services which may be using the same channel.

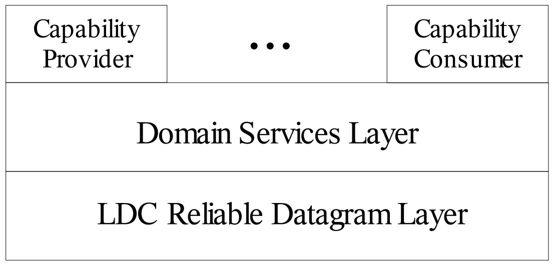
1.2.1 Communication Stack

The domain services mechanism is layered on top of domain channels to facilitate communication between a guest domain and its service entities. The reliable protocol mode of LDC is leveraged to ensure in-order guaranteed packet delivery as well as detection of faults on the communication channel - including loss of connection due to, say, the communication peer crashing or re-booting.

On top of the LDC reliable protocol the DS protocol handles the registration of provider capabilities with their consumer(s), and subsequently the routing of data messages for those registered capabilities.

The content of transported messages is specific to the higher-level protocol between the particular DS service and its client. The DS communication stack is illustrated in figure 1.

Figure 1: Communication Stack



By analogy, just as LDC provides a low level transport, like IP, the domain services protocol provides a name service and connection transport protocol, like TCP, to facilitate communication between a capability provider and its consumer.

Messages for a set of registered capabilities are multiplexed over a shared LDC channel. This basic communication flow is illustrated in Figure 2.

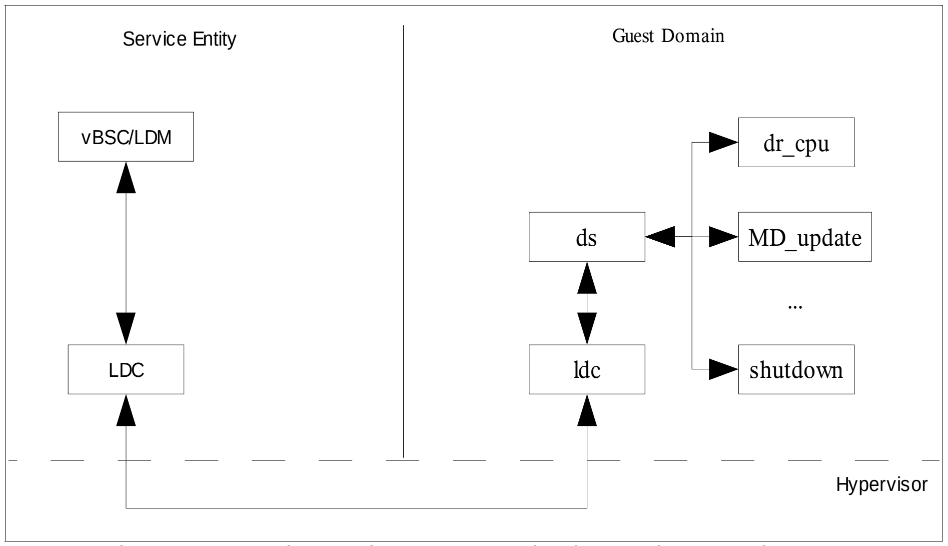


Figure 2: Domain Services Communication Path Example

2 Domain Services Protocol

2.1 Definitions

All fields in this specification are assumed to be in network byte order (big-endian).

2.2 DS Message Header

All DS messages consist of a fixed sized header followed by a variable length data payload. The header format is as follows;

```
typedef struct {
      uint32_t     msg_type;     /* Message type */
      uint32_t     payload_len;  /* Payload length */
} ds_hdr_t;
```

The data payload content is defined according to the msg_type field as follows;

2.3 DS protocol fixed message types

The DS protocol always supports the following three message types and payloads, independent of the current version of the protocol. The type-specific payload is described below each type.

The message types described in this section are intended for version negotiation of the basic DS protocol. All other message types are undefined until the DS protocol version has been negotiated.

The underlying LDC reliable protocol layer will ensure error-free packet delivery, so corrupted packets will already have been dropped. However, receipt of unknown packet types may still occur as a result of bugs or due to malicious guest OS behavior. Upon the receipt of an unknown or undefined (for the currently negotiated DS protocol version) packet type, the recipient should discard the datagram, and close the LDC channel. This action resets the domain services channel connection. Re-opening the channel again should ensure complete end-to-end protocol negotiation and re-registration of capabilities.

2.3.1 Initiate DS connection

```
DS_INIT_REQ         0x0

typedef struct {
      uint16_t     major_vers;
      uint16_t     minor_vers;
} ds_init_req_t;
```

2.3.2 Initiation acknowledgment

```
DS_INIT_ACK         0x1

typedef struct {
      uint16_t     minor_vers;
} ds_init_ack_t;
```

2.3.3 Initiation negative acknowledgment

```
DS_INIT_NACK  0x2

typedef struct {
      uint16_t     major_vers;
} ds_init_nack_t;
```

2.4 DS protocol version negotiation

The DS protocol negotiation involves a countdown algorithm in an attempt to agree on a common major number. Major numbers correspond to incompatible changes; both sides must agree on a major version number for the version negotiation to proceed. As part of agreeing on a major number agreement, each side learns of the other's highest supported corresponding minor number. Minor numbers correspond to back-compatible changes; the two sides implicitly agree to use the lower of the two minor numbers exchanged, and the negotiation is successfully completed.

Specifically, the negotiation is initiated by the guest sending the DS_INIT_REQ message to the service entity listening on the other end of the domain channel. This message includes major and minor version numbers supported by the guest. If the service entity can't support the major version number sent from the guest, it responds with the DS_INIT_NACK message, specifying the closest major version number it can support. The guest can then initiate a new negotiation if it wants (i.e. if it can support the alternate major number returned by the service entity). However, if the service entity's DS_INIT_NACK message includes a major number of zero, the service entity should assume that the guest does not support any version of the DS protocol in common with it.

2.5 DS protocol version 1.0

2.5.1 DS Message types defined for v.1.0 of the DS protocol

2.5.1.1 Register Service

```
DS_REG_REQ          0x3

typedef struct {
      uint64_t     svc_handle;
      uint16_t     major_vers;
      uint16_t     minor_vers;
      char         svc_id[];     /* Up to MAX_STR_LEN */
} ds_reg_req_t;

MAX_STR_LEN         1024         /* MAXPATHLEN */
```

2.5.1.2 Register Acknowledgment

```
DS_REG_ACK          0x4

typedef struct {
      uint64_t     svc_handle;
      uint16_t     minor_vers;
} ds_reg_ack_t;
```

2.5.1.3 Register Failed

```
DS_REG_NACK         0x5

typedef struct {
      uint64_t      status;
      uint64_t      svc_handle;
      uint16_t      major_vers;
} ds_reg_nack_t;
```

A DS_REG_NACK message can return the following status codes:

```
DS_OK        0x0   Success
DS_REG_NACK  0x1   Cannot support requested major version
DS_REG_DUP   0x2   Duplicate registration attempted
```

2.5.1.4 Unregister Service

```
DS_UNREG            0x6

typedef struct {
      uint64_t      svc_handle;
} ds_unreg_req_t;
```

2.5.1.5 Unregister OK

```
DS_UNREG_ACK  0x7

typedef struct {
      uint64_t      svc_handle;
} ds_unreg_ack_t;
```

2.5.1.6 Unregister Failed

```
DS_UNREG_NACK 0x8

typedef struct {
      uint64_t      svc_handle;
} ds_unreg_nack_t;
```

2.5.1.7 Data Message

```
DS_DATA             0x9

typedef struct {
      uint64_t      svc_handle;
} ds_data_handle_t;
```

Note: The ds_data_handle_t header is defined so that when combined with the basic DS header the final payload delivered a service is aligned on a 64bit boundary with regard to the entire DS datagram delivered by LDC.

This alignment is to enable an implementation to potentially utilize an optimized copy when/if creating a message buffer for the final destination service.

2.5.2 Service Handles

A service handle is an opaque 64 bit descriptor that uniquely identifies an instance of a service. It is analogous to a TCP port number, and is specified as part of the DS_REG_REQ message payload, sent to begin the negotiation/registration process for a capability. It is used during this phase to identify the specific negotiation in progress (there could be more than one). Once a capability has been registered, it is used to identify the entity to be notified on receipt of a message. Similarly, when a capability sends a message to a client, the handle identifies the sender. It also identifies the target service during the unregistration process.

2.5.3 Service Identifier

The DS_REG_REQ message specifies a Service Identifier (svc_id), a NULL-terminated character string naming the service. The format and restrictions on the svc_id string are identical to the PROP_STR type's data field as defined in the Machine Description Specification [md].

2.5.4 DS Capability Version Negotiation & Registration

Version negotiation for DS capabilities utilizes exactly the same countdown algorithm as used in the DS Protocol version negotiation, with the same semantics for major & minor numbers, and corresponding message types for implementation. The details of that portion of the protocol are not repeated here.

The registration process is the way in which DS capabilities advertise their availability. A registration is initiated by the service sending a DS_REG_REQ message containing both a service handle and a service identifier.

In response to a successful registration, the other side sends back a DS_REG_ACK message that includes the same service handle provided in the original message. Until this response is received, the DS service interface for this client is not available.

This negotiation/registration handshake must occur whenever the underlying LDC comes up. If there is an event that causes the LDC to go down, all services are automatically unregistered. When the channel comes back up, all services must therefore re-register themselves.

2.5.5 Service Requests

Once the registration handshake has occurred, a DS client can send data messages to any of the registered servers by sending a DS_DATA message.

The data message payload includes the 'ds_handle' of the service that is the intended recipient of the message. Following that is any service-specfic payload; the 'payload_len' field of the header is the length of the entire payload.

The final recipient of the message payload does not receive the DS header or the ds_handle. It only receives the remainder of the payload and an indication of the length of that portion of the payload.

If there is an error in the message that results in the inability of DS to forward the message to the intended recipient, a reply message is sent back with an error in the status field. Note that the original payload is not returned.

If the message is forwarded all the way to the service successfully, the higher level protocol implemented by that service determines what if any reply message is sent.

2.5.6 Unregistration

In the event that a capability becomes unavailable, such as if the kernel module that provides it is unloaded, a DS_UNREG message is sent.

The 'ds_handle' field of the DS header is filled with the service handle that uniquely identifies the registered service. There is no payload to this message.

Once the first message is received, the service handle is invalidated and connections to that service are closed.

If the DS LDC channel goes down, all registered services are forced to the unregistered state by one or both sides that are still running. Before a service can be used again, both the DS infrastructure handshake and the service registration handshake must be re-negotiated.

Service handles should not be reused after a service is unregistered. This prevents successful use of a stale handle. Service handles may be re-used after the basic LDC connection is taken down and then up, and the overall DS framework is reset as a result.

3 DS Capabilities

A *DS capability* is defined as any service provided by one subsystem on behalf of another. Capabilities are based on functionality rather than software module boundaries. Thus, a module can register multiple capabilities if it provides multiple features that are logically grouped together. Associated with a capability are a service identifier and a service handle.

The following sections describe the core DS capabilities supported in a Logical Domain environment.

3.1 MD Update Notification version 1.0

The MD update capability allows a service entity to notify a guest when the entity has modified the guest's Machine Description. It is the responsibility of the MD update capability to parse the new MD, determine what has changed, and initiate the steps required to adjust the guest configuration accordingly. The exact steps taken upon receiving an MD update notification may vary depending on the type of guest running in the domain.

3.1.1 MD Update Request

```
typedef struct {
      uint32_t     seqno;
} ds_md_update_req_t;
```

3.1.2 MD Update Response

```
typedef struct {
      uint64_t     status;
} ds_md_update_resp_t;

/* MD update status */
MD_UPDATE_SUCCESS          0x1
MD_UPDATE_FAILURE          0x2
MD_UPDATE_INVALID_MSG      0x3
```

3.2 Domain Shutdown version 1.0

The Domain Shutdown capability allows a service entity to send a DS_DATA message requesting a guest to gracefully shutdown. A time delay can be specified (in mS).

3.2.1 Domain Shutdown Request

```
typedef struct {
      uint32_t     seqno;
      uint32_t     ms_delay;
} ds_domain_shutdown_req_t;
```

3.2.2 Domain Shutdown Response

```
typedef struct {
      uint64_t     status;
      char         reason[];    /* Optional; less than
                                   MAX_SR_LEN */
} ds_domain_shutdown_resp_t;

/* Domain shutdown status */
DOMAIN_SHUTDOWN_SUCCESS          0x1
DOMAIN_SHUTDOWN_FAILURE          0x2
DOMAIN_SHUTDOWN_INVALID_MSG      0x3
```

3.3 Domain Panic version 1.0

The Domain Panic capability allows a service entity to send a DS_DATA message requesting a guest to panic and cause a crash dump to be created.

3.3.1 Domain Panic Request

```
typedef struct {
      uint32_t     seqno;
} ds_domain_panic_req_t;
```

3.3.2 Domain Panic Response

```
typedef struct {
      uint64_t     status;
      char         reason[];    /* Optional; less than
MAX_SR_LEN */
} ds_domain_panic_resp_t;

/* Domain panic status */
DOMAIN_PANIC_SUCCESS             0x1
DOMAIN_PANIC_FAILURE             0x2
DOMAIN_PANIC_INVALID_MSG   0x3
```

Appendix A: Capability Table

This table lists the capabilities described in this document, and which need to be added to a Domain Services registry.

Service ID	Description
md_update	Notification of MD updates
domain_shutdown	Request graceful shutdown
domain_panic	Request a panic

Appendix B: References

[ldc]	Logical Domain Virtual IO Architecture Specification [Awaiting FWARC case number]
[md]	Machine Description Specification FWARC/2005/115