Simulated Railroad Framework, http://simulrr.sourceforge.net
Synopsis: 100_SrrFramework

This file valid for step 0033.10.5
Issue Date: 2019-06-09

# The SMS Base

## 1 Synopsis

This paper describes a few very basic elements of the SRR/SMUOS Framework.

- The concept of basic initialization
- The SMS Loader.....in the file sms/SmsLoader.x3d
- The Module Wrappers.....in the files sms/SmsModuleWrapper.x3d and sms/SmsModuleWrapperDependent.x3d
- The SMS Tracer.....in the file sms/SmsTracer.x3d
- The UBO Wrapper.....in the file sms/SmsUboWrapper.x3d
- The Little Loader.MOB.....in the files sms/SmuosXMobMyLittleLoader.x3d and sms/SmuosXMobMyLittleLoaderNs.x3d

## 2 Basic Initialization

When you do your first look into some X3D prototypes of the SRR/SMUOS Framework, then you might immediately recognize there are four mysterious fields at the beginning of each <ProtoInterface>:

```
<ProtoDeclare name='McCore'>
    <ProtoInterface>
        <!--  Common fields for the MASTER/DEP state machine  -->
        <field accessType='outputOnly' name='sendLoaded' type='SFBool'/>
        <field accessType='inputOnly' name='receivePing' type='SFBool'/>
        <field accessType='outputOnly' name='sendPong' type='SFBool'/>
        <field accessType='inputOnly' name='receiveBasicInit' type='SFBool'/>
        <!--  Specific fields for McCore  -->
        ........
        ........
```

These fields serve the "basic initialization" with the MASTER/DEP state machine.

Basic initialization is a means to wait with the processing, until all external prototypes have been loaded for a scene. Thus we can avoid loosing events by waiting until basic initialization, before we start sending events.

Each scene that contains at least one <ProtoInstance> node that points to an external prototype, contains a MASTER script, which uses that four fields of each <ProtoInstance> to wait with the basic initialization, until all external prototypes have been completely loaded.

Please find a description of the concept in Appendix A.

## 3 The SMS Loader

The SMS Loader is a prototype, which is used
- by the SSC Base to load/unload module related SSC Dispatchers
- by the UBO Loader to load/unload UBOs (*not yet implemented*)
- by the Little Loader MOB to load/unload dynamic modules or models

The SMS Loader is not currently described in detail.

## 4 The Module Wrappers

Currently we have got a module wrapper according to the "VRML paradigm", where a top level module is wrapped by a <Transform> node that applies a rotation around the y-axis and an arbitrary translation. Scaling of modules is not currently foreseen.

Each module wrapper implements following main functions:

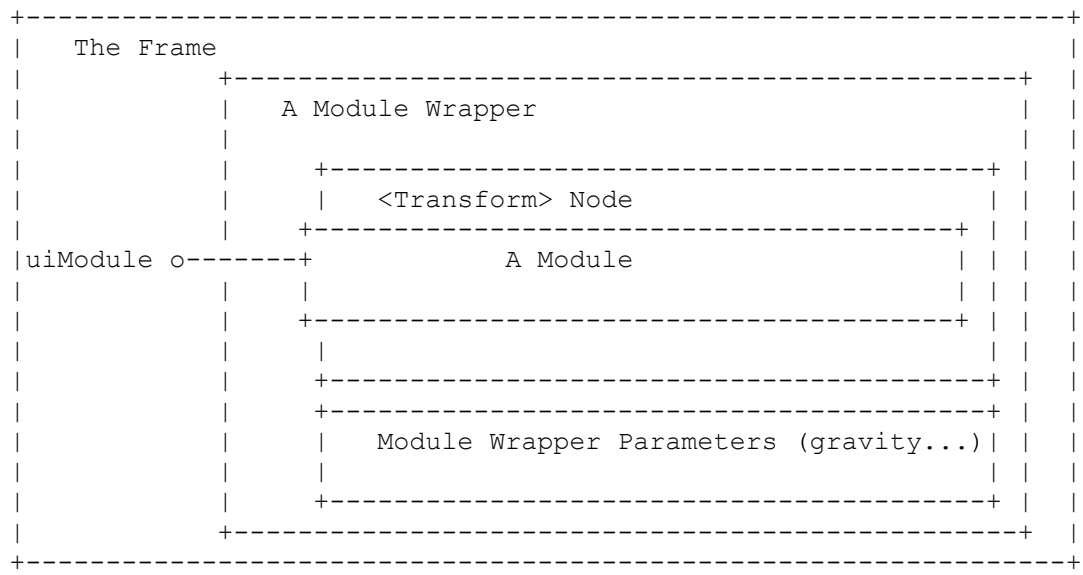
- collecting
  - module name ("moduleName" (SFString)),
  - either URLs or commParam ("url" (MFString) or "commParam" (SFNode))
  - mwParam ("mwParam" (SFNode)),
  - translation ("translation" SFVec3f),
  - rotation ("rotation" SFRotation)

  before loading the module (with "commParam" or "url" whichever is the last)
- loading the module by Browser.createVrmlFromURL()
- setting the initial state of the module from the input during load
- performing the basic initialization and the initialization of the module
- "adding" the module to the transformation

The module can be loaded again after unloading. The module wrapper is supposed to remain in the memory as long as the module is registered at the SSC.

Each module wrapper contains a <Transform> node, where the module shall be "hung up", and it contains the script with the "module wrapper parameters".

The "module wrapper parameters" contain the field "requiredMcExtensions" (MFString) – which is currently empty – and the field "gravity" (SFVec3f), which is currently a constant gravity of 9,81 m/s^2 in -y – direction.




*TODO11* this chapter must be updated, when the dependent modules are realized.

Note: Having the module wrappers in memory independently of the actual presence of the module, is a preparation for some days, when dynamic modules – that have been registered but NOT loaded – must contribute to the gravity.

Note: Currently the main duty of module wrappers is to "wrap modules by some transformation". "Caring for mass and gravity as well as acceleration" could become a duty of the module wrappers in a later release.

Note: So we have a concept of module "position" and "orientation", but none of "velocity", nor one of "acceleration", currently.

# 5 The SMS Tracer

## 5.1 Purpose of the SMS Tracer

It is the intention to use the SMS Tracer for
- getting familiar with SRR/SMUOS software
- debugging SRR/SMUOS software
- documenting SRR/SMUOS software

The programmer can write diagnostic output to the tracer, which will be
- output to the browser's console
- output at the uiControl interface

Output via uiControl is intended to enable processing tracer output by external programs via the SAI/EAI.

Each place in the code, where diagnostic output is sent to the tracer - we call these places tracepoints - has assigned a trace level.

Only if the actual trace level of the system is equal to or greater than the trace level of the tracepoint, then the diagnostic output will actually be written.

Different parts of the system can have different trace levels, enabling narrowing down the error/effect you're looking for.

### 5.1.1 Trace Levels

0.....no tracer output
1.....Errors (this trace level is the default setting in the system)
2.....Infos
3.....Debug Infos

Trace levels are set locally in one scene instance and must hence be set in each required scene instance separately.

The trace levels can be set at the uiControl interface with the following input fields of the Simple Scene Controller.

### 5.1.2 "traceLevelRequest" (SFInt32)

This input field sets the "classic" trace level. The "classic" tracer is not used by the SRR/SMUOS Framework and is hence not described here.

### 5.1.3 "traceLevelSscBaseRequest" (MFInt32)

This input field sets the trace level of the SSC Base (i.e. of the client of the SSC Base and of the clients of some SSC extensions that follow SSC Base).

The input field reacts on the first two elements of the MFInt32 array, the trace level [0] will be applied AFTER initialization ("operational" trace level) and the trace level [1] will be used during initialization ("initialization" trace level).

The switchover between the two trace levels happens, when the SSC Base issues the "common parameters" (commParam), i.e. BEFORE the module coordinators are initialized.

### 5.1.4 "traceLevelCommControlRequest" (MFInt32)

This input field sets the trace level of the server of the SSC Base and of the servers of some SSC extensions that follow the SSC Base.

The input field reacts on the first two elements of the MFInt32 array, the trace level [0] will be applied AFTER initialization ("operational" trace level) and the trace level [1] will be used during initialization ("initialization" trace level).

The switchover from "initialization" trace level to "operational" trace level happens, when the SSC Base issues the "common parameters" (commParam), i.e. BEFORE the module coordinators are initialized.

When you want to trace the server of the SSC Base, be sure to have the "central controller role" in your scene instance!

### 5.1.5 "traceLevelModulesRequest" (SFString)

This SFString value requests the trace levels for all(!) modules in a simple syntax.

Just request
   <moduleName>=<tloper>[,<tlinit>][;<moduleName>=<tloper>[,<tlinit>]]...

for as many modules as you like, where
   <moduleName> is the name of the module in question ('*' as a place holder is allowed)
   <tloper> is the "operational" trace level
   <tlinit> is the "initialization" trace level.

If you omit <tlinit>, it will be set to <tloper>.

The switch over from "initialization" trace level to "operational" trace level happens, when the module coordinator issues the "module parameters" (modParam), i.e. BEFORE the MIDAS Objects are initialized.

The SRR/SMUOS Framework will automatically add a leading term "*=1,1;" to set the trace level of all modules that you do not specify.

### 5.1.6 "traceLevelObjectsRequest" (SFString)

This SFString value requests the trace levels for all(!) objects in a simple syntax.

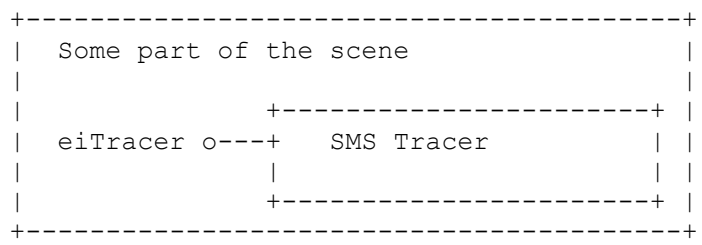
The syntax and logic is similar to (3.1.5), but <dispatcherName>-<objId> is used instead of <moduleName> and only one trace level is used (<tlover>, the "overall" trace level").

When you want to trace the <dispatcherName>-<objId>.ObCo instance of a MIDAS Object, be sure to have the "MOC role" for the parent/current module in your scene instance!

## 5.2 External View of the Tracer

The previous chapter told us, how the tracer can be used by the gamer, when he sets the various trace levels of the Simple Multiuser Scene.

The following sections will tell us, how the programmers of MIDAS Objects, SSC Extensions and MC Extensions and the authors of Models, Modules and Frames can use the tracer via its external interface eiTracer.




Where "Some part of the scene" can be
- either a MIDAS Object or
- an SSC Extension or
- an MC Extension or
- a Model or
- a Module or
- the Frame.

- eiTracer ..... is the external interface of the SMS Tracer prototype

### 5.2.1 eiTracer – The External Interface of the SMS Tracer Prototype

The following table shows the use cases of the tracer that are available in each state of the tracer. The tracer can be either initialized or un-initialized.

| Un-initialized | initialized |
|---|---|
| Initialization | Initialization |
| | Setting the Trace Level |
| | Output L1 Errors |
| | Output L2 Infos |
| | Output L3 Debug Infos |

#### 5.2.1.1 Initialization of a Tracer Instance

The SMS Tracer needs the pointer to the "Common Parameters", because it uses some basic services of the "Common Parameters" to output the tracer output to the Web3D browser's console and to the uiControl interface of the Simple Scene Controller.

Some "Tracing Cases" need the pointer to the "Module Parameters" as an additional information, whether the tracer is contained in a module or not.

Hence the SMS Tracer can be initialized or re-initialized either by "commParam" or by "modParam". Changing between both kinds of being initialized is possible.

General Fields at the eiTracer Interface
---

- "subsystem" / "ssVersion" / "fileName" ..... identify the software that instantiates the tracer and that uses the tracer to output the L1, L2 or L3 info
- "instanceId" ..... identifies the instance of the software that instantiates the tracer. Please refer to 011_NamingRules for more information about software instances (VLFs)

The "Tracing Cases"
---

| Tracing Case | universalObjectClass | objId | modParam |
|---|---|---|---|
| Frame | - | - | - |
| Astral Object | - | X | - |
| Module | - | - | X |
| Bound Object | - | X | X |
| Forbidden | X | - | X/- |
| Unbound Object | X | X | X/- |
| SSC Parm. Object | X | X | - |

Following output will be present in addition to the concrete output of the trace point (which is set by the programmer):

| Tracing Case | instanceId | objId | moduleName | subsystem/fileName/ssVersion |
|---|---|---|---|---|
| Frame | X | - | - | X |
| Astral Object | X | X | - | X |
| Module | X | - | X | X |
| Bound Object | X | X | X | X |
| Forbidden | - | - | - | - |
| Unbound Object | X | X | X/- | X |
| SSC Parm. Object | X | X | - | X |

#### 5.2.1.2 Setting the Trace Level of a Tracer Instance

The field "localTraceLevel" (SFInt32) can be used to set the local trace level of THIS tracer instance.

This is possible for the "Tracing Cases"
- Frame and
- Module

The other "Tracing Cases", which are about objects and models, namely
- Astral Object
- Bound Object
- Unbound Object
- SSC Parm. Object

take their trace levels automatically from the "commParam.traceLevelObjects" field and report the resulting trace level at the "localTraceLevel" field.

#### 5.2.1.3 Output L1 Errors via a Tracer Instance

The eiTracer field "errLog" (SFInt32) can be used to set the overall "errorNo" parameter in the "commParam" and to output the corresponding "L1 Error Message".

The programmer cannot influence the "L1 Error Messages", which are predefined in the Common Parameters. You can search for "errorStrings" in prototype "SscBase".

#### 5.2.1.4 Output L2 Infos via a Tracer Instance

The eiTracer fields "freeTextInfo" (MFString), "messageReceived" (MFString), "sendMessage" (MFString), "eventReceived" (MFString), "sendEvent" (MFString), "newState" (MFString), "startTimer" (MFString), "stopTimer" (MFString), "timerExpired" (MFString), "instanceStarted" (MFString) and "instanceStopped" (MFString) can be used to output L2 Info via the tracer.

The field "freeTextInfo" just outputs "L2 Info Text", as it was given by the programmer of the trace point.

The other fields interpret the first string of the MFString array in a specific way and output the other strings additionally (as an "L2 Info Text"). The first string is split into parameters, which are separated by commas.

```
"messageReceived" ........ firstString = "<sender>,<messageType>,<messageId>"
"sendMessage" ............ firstString = "<receiver>,<messageType>,<messageId>"
"eventReceived" .......... firstString = "<origin>,<field>"
"sendEvent" .............. firstString = "<destination>,<field>"
"newState" ............... firstString = "<stateId>"
"startTimer" ............. firstString = "<timerId>"
"stopTimer" .............. firstString = "<timerId>"
"timerExpired" ........... firstString = "<timerId>"
"instanceStarted" ........ firstString = "<instanceId>"
"instanceStopped" ........ firstString = "<instanceId>"
```

Example:

```
tracerControl.eventReceived = new MFString(
           extObjId + '@uiObj,bindBeamerDestination',
          'value=' + Value,
          'user requests to bind beamer destination');
```

The above call of the tracer outputs a trace point, after an event has been received,

- where the event's "origin" is extObjId + "@uiObj" and
- the event's "field" is "bindBeamerDestination"

Two additional lines of "L2 Info Text" will be output,

- one with "value=" + Value and
- one with "user requests to bind beamer destination".

#### 5.2.1.5 Output L3 Debug Infos via a Tracer Instance

The eiTracer field "freeTextDebug" (MFString) can be used to output L3 Debug Info via the tracer. It just outputs the "L3 Debug Info Text", as it was given by the programmer of the trace point.

# 6 The UBO Wrapper

Tbd.

*TODO11*  this chapter must be updated, when the UBOs are realized.

## 7 The Little Loader MOB

The SMUOS Framework provides the prototypes "MyLittleLoader" and "MyLittleLoaderNs", which can be used

- as loader for dynamic modules (DynMos) .....initialized (MOO I)
- as loader for dynamic bound objects (DynBos) .....attached (MOO II)

### 7.1 If Initialized in MOO I, the Little Loader MOB Operates as Loader for DynMos

After initialization, the Module Loader reads the "Dynamic Module Configuration" (Dmc) from commParam, where they have been stored by the SSC Base. Actually they are contained in the "Dynamic Element Description" (DED), which holds information about ALL dynamic elements.

The DED contain among others following fields with the description of all dynamic modules:

- the following fields are processed by the Little Loader MOB (standard)
  - moduleNames (MFString) ..... names of all dynamic modules
  - moduleUrls (MFString) ..... URLs of all dynamic modules
- the following field is processed by the module wrapper (content depends on the type of the wrapper and is encoded by name=value, semicolon-separated)
  - moduleParameters (MFString)... e.g. translation and rotation of module
- the following fields are specific to the SrrTrains demo layout, they are processed directly by the main file
  - proxiCenters (MFVec3f) ..... center points of the proxi sensors
  - proxiSizes (MFVec3f) ..... sizes of the proxi sensors

We see, the Little Loader MOB supports the author with some basic topics of dynamic modules, e.g. registration, deregistration, loading, unloading.

However, the WHY the module is to be loaded or unloaded, e.g a trigger by some proxi sensor, has still to be implemented by the author.

#### 7.1.1 The Module Loader provides the following fields on its external interface

- <field accessType='inputOutput' name='supportedSmuosExtensions' type='MFString'/>

The field "supportedSmuosExtensions" must list all well-known-IDs of all SSC Base Extensions that shall be supported by the present instance of the Loader.

How can we understand this? Each "moduleName" in the DED is prefixed with the WKI of the SSC Extension that must be present to support the module wrapper (if any). Now the Module Loader searches for this SSC Extension (if any), where it is referred to the URL of the Module Wrapper that must be loaded. Otherwise it looks for the URL of the default Module Wrapper at the SSC Base.

- <field accessType='outputOnly' name='registerModules' type='MFString'/>
- <field accessType='inputOutput' name='registeredModules' type='MFString'/>

The fields "registerModules" and "registeredModules" are connected to the fields of the same name of the SSC Base. Once the SSC is initialized, it reports the commParam. With the commParam now also the Module Loader is initialized, reads the contents of the DED and registers the dynamic modules by "registerModules". Throughout the simulation, the SSC keeps the list of registered modules up to date in the "registeredModules" field, which contains both dynamic and static modules.

```
<field accessType='inputOutput' name='dynMods' type='MFNode'/>
<field accessType='outputOnly' name='moduleWrapperUnloaded' type='SFInt32'/>
<field accessType='inputOutput' name='moduleWrapperParameters' type='SFString'/>
<field accessType='outputOnly' name='moduleWrapperLoaded' type='SFInt32'/>
<field accessType='inputOnly' name='loadModule' type='SFInt32'/>
<field accessType='inputOnly' name='unloadModule' type='SFInt32'/>
```

As soon as the module wrapper has been loaded (which is done immediately after registration), the Module Loader logs in with the field "moduleWrapperLoaded". The value of this event points to the module wrapper in the "dynMods" field and allows the frame to set the proprietary parameters of the module from "moduleWrapperParameters".

### 7.1.2 Loading a dynamic module

If the frame decides to load a registered dynamic module, then it must pass the moduleIx to the Module Loader in the "loadModule" field.

This will ensure that the module, if it was already loaded, will be initially unloaded and then freshly loaded.

Now the module is actually being loaded and initialized.

### 7.1.3 Disabling and unloading a dynamic module

The index of a module (the so-called moduleIx) is the index at which the SSC outputs the module name in the field "registeredModules".

Now, if the frame wants to unload a dynamic module, it must pass the module's moduleIx to the "unloadModule" field.

Furthermore, the Module Loader automatically unloads a dynamic module, if it has been loaded and when the SSC deletes the module name from the "registeredModules" field (if it is being deregistered).

## 7.2 If Attached in MOO II, the Little Loader MOB Operates as Loader for DynBos

*TODO11* this chapter must be updated, when the model loader is realized.

# Appendix A – Basic Initialization

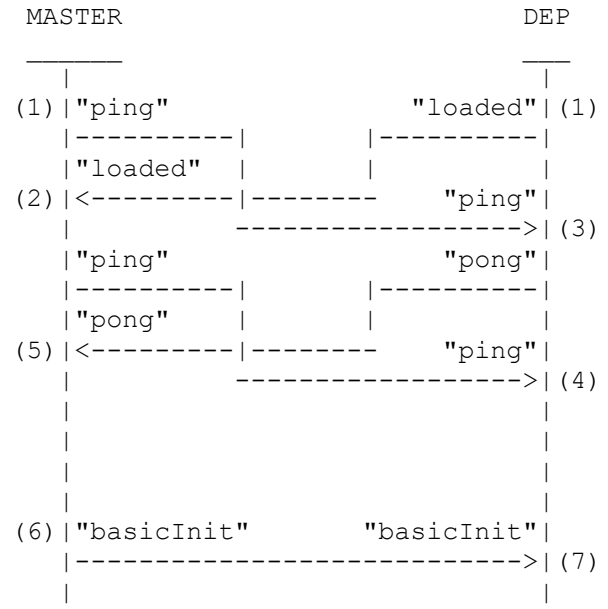
## Concept for MASTER/DEP State Machine ("Load Sensor" for External Prototypes)

Motivation: (a)
VRML/X3D Browsers may load files asynchronously.
I.e., if a file refers to other files, then it may happen that one file finishes loading before the other files finish.
Hence it may happen that events passed from a node of one file to a node of another file may get lost.
This is particularly true for events that are sent from the initialize() function of a Script node over file borders.
(b)
Sometimes, we load parts of the scene dynamically (using the Browser.createVrmlFromURL() method). It may happen that the loaded part of the scene gets initialized, before we insert it to a Group node and before we create dynamic routes to exchange events with the loaded part of the scene. Hence the simple solution of just outputting an event from the loaded part of the scene, as soon as it gets initialized, may fail.

Summary:

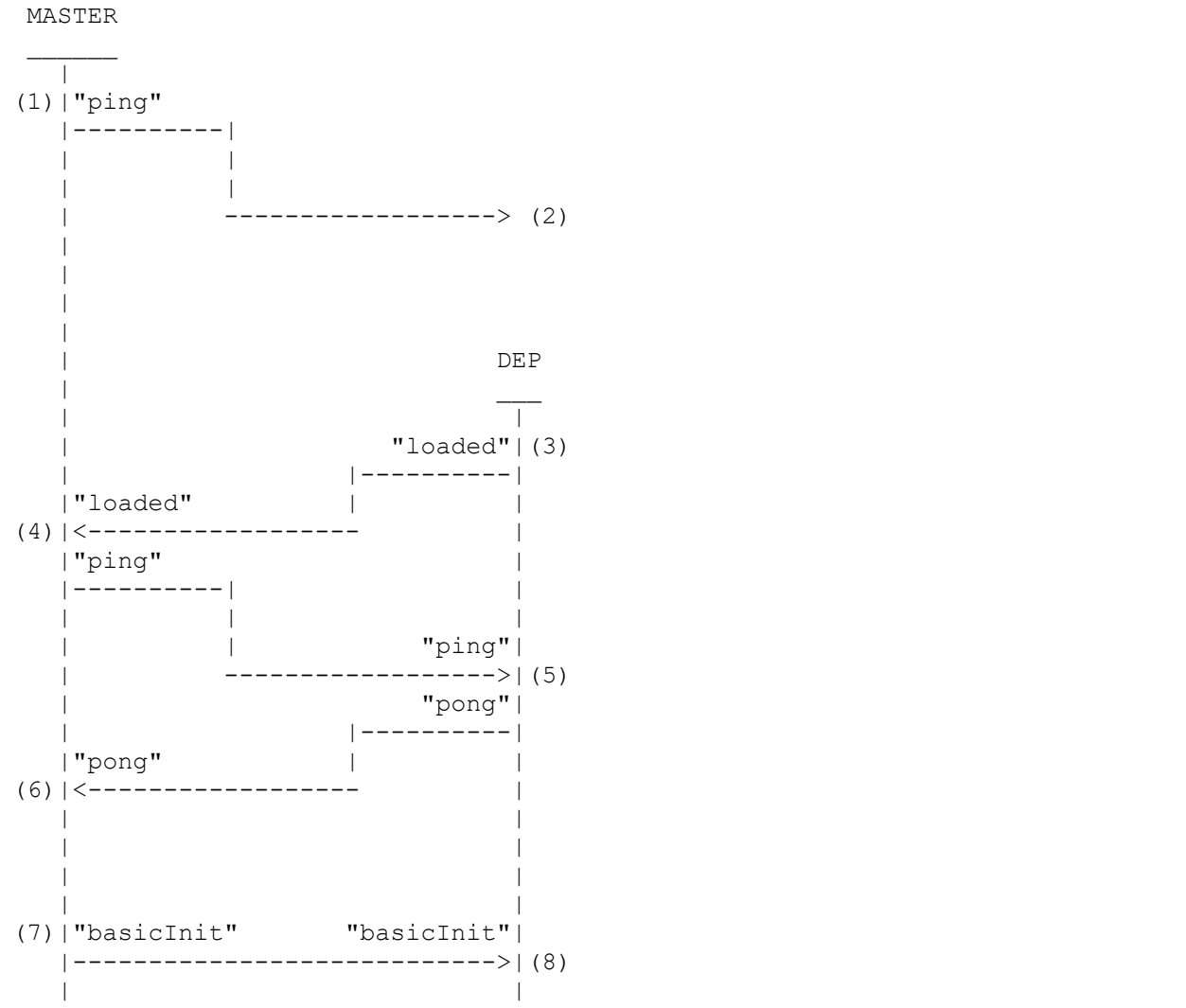
A simple concept is developed, where each external prototype has to contain a "dependent" Script node (DEP) and where the loading file (the file which contains the proto instances) contains a "master" Script node (MASTER) and some routes between the proto instances and the MASTER.
As soon as all external prototypes are loaded, the MASTER distributes a "basicInit" event to all prototypes. Hence the prototypes can exchange events arbitrarily during "basic initialization" without loosing events.
The term "basic initialization" refers to the initialization, which is triggered by the mechanisms of the present concept, it is performed AFTER the "normal Web3D initialization" (initialize()).

Scenario I: MASTER and DEP are loaded and initialized synchronously
==========



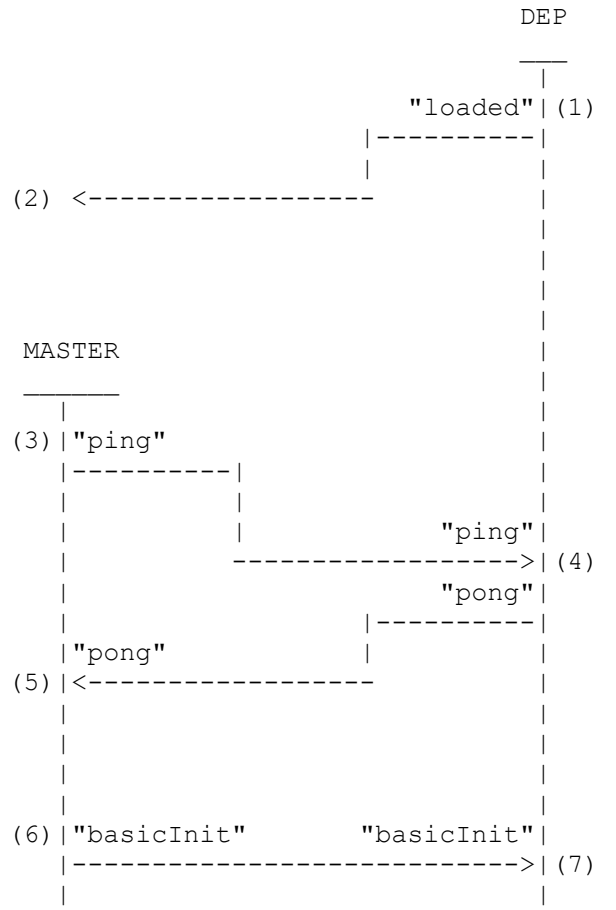

(1)...MASTER and DEP are initialized, send "ping" and "loaded", resp.
(2)...MASTER receives "loaded" and sends another "ping"
(3)...DEP receives "ping" and responds with "pong" (first "ping")
(4)...DEP receives a second "ping" and ignores it
(5)...MASTER receives "pong" and increments DEP counter
(6)...as soon as DEP counter reaches the maximal value (now all DEPs are loaded and initialized), then MASTER sends "basicInit"
(7)...all DEPs are "basically initialized" now, they may exchange events arbitrarily without any event getting lost

Scenario II: MASTER is loaded and initialized first
==========




(1)...MASTER is initialized and sends "ping"
(2)...the first "ping" gets lost
(3)...DEP is initialized and sends "loaded"
(4)...MASTER receives "loaded" and sends another "ping"
(5)...DEP receives "ping" and responds with "pong"
(6)...MASTER receives "pong" and increments DEP counter
(7)...as soon as DEP counter reaches the maximal value (now all DEPs are loaded and initialized), then MASTER sends "basicInit"
(8)...all DEPs are "basically initialized" now, they may exchange events arbitrarily without any event getting lost

Scenario III: DEP is loaded and initialized first
============




(1)...DEP is initialized and sends "loaded"
(2)...the "loaded" gets lost
(3)...MASTER is initialized and sends "ping"
(4)...DEP receives "ping" and responds with "pong"
(5)...MASTER receives "pong" and increments DEP counter
(6)...as soon as DEP counter reaches the maximal value (now all DEPs are loaded and initialized), then MASTER sends "basicInit"
(7)...all DEPs are "basically initialized" now, they may exchange events arbitrarily without any event getting lost

Resulting Description of the Concept
====================================

(A) The loading file contains a Script node "MASTER" and routes between the MASTER and the proto instances
(B) Each proto declare of the external prototypes contains a Script node "DEP"
(C) In case of nested prototypes, the Scripts in the intermediate prototypes take care about the "MASTER duties" and about the "DEP duties"
(D) Each MASTER has an "initializeOnly" "SFInt32" that indicates the number of dependents ("numDeps")
(E) Each MASTER has an "outputOnly" "SFBool" "sendPing"
(F) Each DEP has an inputOnly" "SFBool" "receivePing"
(G) Each DEP has an "outputOnly" "SFBool" "sendLoaded"
(H) Each MASTER has an "inputOnly" "SFBool" "receiveLoaded"
(I) Each DEP has an "outputOnly" "SFBool" "sendPong"
(J) Each MASTER has an "inputOnly" "SFBool" "receivePong"
(K) Each MASTER has an "outputOnly" "SFBool" "sendBasicInit"
(L) Each DEP has an "inputOnly" "SFBool" "receiveBasicInit"
(M) Each MASTER has an "inputOutput" "SFInt32" "depCounter" "0"
(N) Each DEP has an "inputOutput" "SFBool" "ignorePing" "true"
(O) Behaviour of the MASTER

```
function initialize()
{
  if (numDeps)
    sendPing = true;
  else
    sendBasicInit = true;
}
function receiveLoaded()
{
  sendPing = true;
}
function receivePong()
{
  if (depCounter < numDeps)
  {
    if ((++depCounter) >= numDeps)
    {
      sendBasicInit = true;
    }
  }
}
```

(P) Behaviour of the combined MASTER/DEP

```
function initialize()
{
  if (numDeps)
    sendPing = true;
  else
    iAmLoaded();
}
function iAmLoaded()
{
  ignorePing = false;
  sendLoaded = true;
}
function receiveLoaded()
{
  sendPing = true;
}
function receivePing()
{
  if (!ignorePing)
  {
    ignorePing = true;
    sendPong = true;
  }
}
function receivePong()
{
  if (depCounter < numDeps)
  {
    if ((++depCounter) >= numDeps)
    {
      iAmLoaded();
    }
  }
}
function receiveBasicInit()
{
  // TO DO: do my basic initialization here
  sendBasicInit = true;
}
```

(Q) Behaviour of the DEP

```
function initialize()
{
  ignorePing = false;
  sendLoaded = true;
}
function receivePing()
{
  if (!ignorePing)
  {
    ignorePing = true;
    sendPong = true;
  }
}
function receiveBasicInit()
{
  // TO DO: do my basic initialization here
}
```