Data Access Layer Plug Implementation

Project:	**DESY-CSS**
Document Owner:	Igor Križnar
Status:	***Released***
Creation:	2006-04-24 (Igor Kriznar)
Revision:	1.0

Audience

This document is intended for all interested in DataAccessLayer API implementation.

Scope

The intent of this document is to describe how to write plug implementation for DAL most effectively.

Document History

Revision	*Date*	*Author*	*Section*	*Modification*
1.0	2006-04-24	Igor Križnar	All	Created.
1.1	2006-05-11	Igor Križnar	2.4, 2.5, 2.7	Device implementation added.
1.2	2006-05-15	Igor Križnar	3, I.	API updated.
1.3	2006-06-19	Igor Križnar	4, I.	Directory added, API updated.
1.4	2006-06-22	Klemen Žagar	4.1	Directory proposal added.

References

ID	*Author*	*Reference*	*Revision*	*Date*	*Publisher*
1	Igor Križnar	Data Access Layer API Proposal	1.1	2006-04-18	Cosylab
2	Gašper Tkačik	Data Access Layer Feature Requests http://epics-office.desy.de/content/e14/e193/e194/e196/CSFR.pdf	-	-	Cosylab
3	Sun	Java 5 Generics http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html	-	-	Sun
4	Sun	Java Naming and Directory Interfaces http://java.sun.com/products/jndi/	-	-	Sun
5	EPICS Community	EPICS V4 Name Server http://aps.anl.gov/epics/wiki/index.php/V4_Name_Server	-	-	ANL - ASP
6	DeDiSys	Dependable Distributed Systems (DeDiSys) http://www.dedisys.org	-	-	DeDiSys

ID	Author	Reference	Revision	Date	Publisher
7	T. Howes	The String Representation of LDAP Search Filters http://www.faqs.org/rfcs/rfc2254.html	-	December 1997	Netscape Communications Corp.

Table of Contents

1. Data Access Layer API

Data Access Layer (DAL) prescribes a consistent set of interfaces for access to control system's dynamic and configuration data. More about DAL API is available in document Data Access Layer API Proposal (see ref. 1). We may call an implementation of DAL API a plug. Purpose of plug is to provide glue code between particular control system protocol and DAL API. DAL API uses interfaces rather than abstract classes wherever possible, so the plug may use it ¨s own object or extend already existing control system object in order to keep memory consumption low.

DAL design is object oriented “wide” interface with addition of “narrow“ style generic capabilities. DAL is programmer oriented and objects are “wide”. As consequence they have many methods. Most of the operations can be to accomplished by simply invoking right method on right object, while most of the “inconvenient” stuff is hidden behind API. This approach is friendly to programmer and visual composition editors, but not so friendly to DAL implementors, because they have many methods to implement. Most of the methods are there only for convenience or they are slightly different version of some other method so optimization is possible.

In order to reduce effort necessary to implement DAL API some commong glue code can be defined, which connects DAL API to DAL plug interfaces. Plug API contains

minimum of necessary methods to implement. Glue between simplified plug API and DAL API would consist code, which handles convenience implementation. Most of the glue part has to be implemented by first plug implementor anyway. All subsequent plug implementions can use same common glue code.

This section will explain that there are two ways to write plug for DAL. The first one is to implement whole DAL API from scratch. The other one is to use common plug implementation support classes. Both will be presented in following sections.

1.1 DAL Implementation from Scratch

Most obvious benefit is that implementor has full control over all aspects of implementation and can make best performance optimization. The main disadvantage is that it takes more time and effort to write plug that is DAL API compliant.

1.2 DAL Implementation through Plug Commons

There are several benefits:

- Plug writing is easier, requires less documentation and learning.
- It is easier to ensure DAL compliance.
- It is easier to write simple plug from the beginning with limited set of features working right away and later add full implementation or optimize implementation for performance.
- If the first plug is written consciously with consideration for common classes then minimal additional work is necessary for next plug.

There are also risks, which are listed and described below (with strategies to minimize their influence) :

- Lack of flexibility. This means that common classes allows only one way to implement DAL and if communication layer is not compatible with plug design. Therefor it is hard to implement such plug because it may have problems with performance and stability due to excessive code.
- Heavy common code dragging down performance.

All of these risks can be addressed with following plug commons design requirements:

- Code is thin and minimalistic as possible: better performance and less maintenance.
- Avoiding any heavy frameworks, same reasons as above. Few well choose programming patterns should solve all problems.
- Any common glue code must be replaceable with own code, if circumstances demands it. This enables flexibility. Plug implementation can start as simple extension of common code can be later enhanced to implement special features or performance improvements not provided by commons.

Bottom line is that there is not one way to write plugs but a smooth transition from easier to more complex implementation.

1.3 Other Plug Implementation Aspects

- To ensure DAL API compliance set of JUnit tests should be written and should be used to test any implementations to ensure that it is 100% compatible with DAL design. In fact it is easier to ensure compliance with empiric tests than with written document that is probably never throughly read.
- Simulation is required for several reasons: with simulations DAL concepts can be tested and demonstrated, it can be used for quick testing and demonstrating control system applications offline or before real plug implementation or control system capabilities are even build, which speeds up development process.

2. Plug Implementation on Commons

For testing and demonstration purposes DAL package already contains outline of plug commons and simple implementation of simulation plug. It does not implement all functionality and it focuses only on implementation of dynamic value properties (or channels in EPICS notation), but it does define basic interfaces and design decisions. Plug writing will be explained based on this simple implementation, which will be used as demonstration of concepts.

Asynchronous operations are prefered in plug before synchronous operations. Assumption is that asynchronous operation tend to perform faster, specially when certain tasks must be executed on large amount of properties or remote objects. Another reason is that asynchronous communication is a basic type of communication and it is supported on most communication protocols.

Common plug implementation has three parts:

- DAL API with some default implementation as support classes. Code is in located in following packages: `com.cosylab.datatypes`, `com.cosylab.datatypes.commands`, `com.cosylab.datatypes.context`, `com.cosylab.datatypes.device`, `com.cosylab.datatypes.group`, `com.cosylab.datatypes.spi`.
- Plug API, which in intended to be implemented by plug implementors. Code is in package com.cosylab.datatypes.proxy.
- Common plug code (or glue code). This code is a bridge between plug API and DAL API. This code in intended to be thin and contains stubs of main interfaces declared in DAL API. At the moment this part is not finished. Non-trivial part of common plug code is still missing but the missing parts are well defined on one side with DAL API and on the other with common plug classes API.

2.1 Simulation Implementation

For purposes of this document and testing of design simple simulation implementation has already been written. Simulation implementation is located in package com.cosylab.datatypes.simulation and consist of minimal implementation necessary for DAL to work. Code snippets from simulation will be used further in document as illustration of recommended DAL implementation.

2.2 Implementation Entry Points

Table of interfaces and classes that must be implemented if programmer wants to

implement DAL properties with minimal effort can be found below.

Class or Interface	*Description*
`com.cosylab.datatypes.proxy.PropertyProxy`	Implement this interface to provide connection to communication layer.
`com.cosylab.datatypes.proxy.DeviceProxy`	Implement this interface if your system provides device representation or you want to simulate devices on logical level.
`com.cosylab.datatypes.proxy.DirectoryProxy`	Implement this interface to provide introspection data for property.
`com.cosylab.datatypes.proxy.AbstractPlug`	Extend this abstract class to provide creation of Proxy classes.
`com.cosylab.datatypes.spi.PropertyFactory`	Implement this interface that creates instances of dynamic value properties and binds them with Proxy classes from plug.

Each interface is in more detail explained in following sections.

2.3 Property Implementation

PropertyProxy and DirectoryProxy interfaces are representing connection to communication layer for single DynamicValueProperty. Both interfaces can be implemented with same or with different classes. Class that implements PropertyProxy represents single connection to remote property object. Each of proxies belongs to one uniqe remote name and all methods and requests are relative to this name or remote property.

For each DynamicValueProperty interface there must be corresponding property proxy implementation. Best is to implement as many different property proxies as there are different connection types in communication layer.

Full interfaces are in appendix I. List of methods with sample simulation code as implementation suggestion can be found below.

2.3.1 Proxy Methods

This is common proxy interface, which is shared between PropertyProxy, DeviceProxy and DirectoryProxy.

Method
`public String getUniqueName();`
Explanation
Unique name is name of remote property or data channel.
Code sample from Simulation

`public String getUniqueName() {`
`  return name;`
`}`
where name is obtained from constructor.

Method
`public void destroy();`

Method
Explanation
If during proxy creation any remote resource or connection has been made, now is the time to destroy it. This method can be called only from plug object.
Code sample from Simulation
N/A

Method
`public void addProxyListener(ProxyListener l);` `public void removeProxyListener(ProxyListener l);`
Explanation
Listener should receive event, whenever connection or condition status of proxy changes.
Code sample from Simulation

```
	public void addProxyListener(ProxyListener l) {
		if (proxyListeners==null) {
			proxyListeners= new ListenerList(ProxyListener.class);
		}
		proxyListeners.add(l);
	}
```

Method
`public ConnectionState getConnectionState();`
Explanation
Connection state signals condition of connection to remote object.
Code sample from Simulation

```
	public ConnectionState getConnectionState() {
		return connectionState;
	}
	protected void setConnectionState(ConnectionState s) {
		connectionState=s;
		fireConnectionState();
	}
	protected void fireConnectionState() {
		ProxyListener[] l= (ProxyListener[])proxyListeners.toArray();
		for (int i = 0; i < l.length; i++) {
			try {
				l[i].connectionStateChange(this,connectionState);
			} catch (Exception e) {
				e.printStackTrace();
			}
		}
	}
```

2.3.2 PropertyProxy Methods

Method

```
	public Request getValueAsync(ResponseListener callback) throws
DataExchangeException;
	public Request setValueAsync(T value, ResponseListener callback) throws
```

Method

```
DataExchangeException;
```

Explanation

Asynchronous set and get of remote value. This is basic access to value. ResponseListener implementation is callback, which receives response objects. Each response must have reference to request object (same as returned by these methods), which initiated the response.

Code sample from Simulation

```
      public Request getValueAsync(ResponseListener callback)
                  throws DataExchangeException {
            RequestImpl r= new RequestImpl(this,callback);
            r.addResponse(new
ResponseImpl(this,r,value,"value",true,null,condition,true));
            return r;
      }
      public Request setValueAsync(T value, ResponseListener callback)
                  throws DataExchangeException {
            setValueSync(value);
            RequestImpl r= new RequestImpl(this,callback);
            r.addResponse(new ResponseImpl(this,r,value,"",true,null,condition,true));
            return r;
      }
```

Where com.cosylab.datatypes.impl.RequestImpl and com.cosylab.datatypes.impl.ResponseImpl are default implementations of Request and Response interfaces. Fancier version of simulator would use different thread to return response to callback (by calling r.addResponse(...)).

Method

```
      public boolean isSettable();
```

Explanation

Returns true if remote property accepts set.

Code sample from Simulation

It is always true in simulations.

Method

```
      public MonitorProxy createMonitor(ResponseListener callback) throws
RemoteException;
```

Explanation

This is how value subscription is made to proxy. Implementation should create new instance of MonitorProxy and callback should receive value events. More about this will be explained with MonitorProxy.

Code sample from Simulation

```
      public MonitorProxy createMonitor(ResponseListener callback)
                  throws RemoteException {
            MonitorProxyImpl m= new MonitorProxyImpl(this,callback);
            monitors.add(m);
            return m;
      }
```

Where MonitorProxyImpl is simulated monitor proxy. Will be explained later on.

Method

```
public EnumSet<DynamicValueState> getCondition();
```

Explanation

Condition enum is set of distinctive states, which informs users about remote value quality (such as alarms, timeouts, etc.).

Code sample from Simulation

```
public EnumSet<DynamicValueState> getCondition() {
	return condition;
}
protected void setCondition(EnumSet<DynamicValueState> s) {
	condition=s;
	fireCondition();
}
protected void fireCondition() {
	ProxyListener[] l= (ProxyListener[])proxyListeners.toArray();

	for (int i = 0; i < l.length; i++) {
		try {
			l[i].dynamicValueConditionChange(this,condition);
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

2.3.3 SyncPropertyProxy Methods

This interface is optional. PropertyProxy implementation may decide to implement this interface as well. Basic value access is asynchronous. If this interface is not implemented, then common plug properties asynchronous versions of methods automatically.

Method

```
public T getValueSync() throws DataExchangeException;
public void setValueSync(T value) throws DataExchangeException;
```

Explanation

Synchronous set and get of remote value.

Code sample from Simulation

```
public T getValueSync() {
	return value;
}
public void setValueSync(T value) {
	this.value = value;
	MonitorProxyImpl[] m= monitors.toArray(new
MonitorProxyImpl[monitors.size()]);
	for (int i = 0; i < m.length; i++) {
		m[i].fireValueChange();
	}
}
```

Use of monitor proxy will be explained later.

2.3.4 MonitorProxy Methods

MonitorProxy implementation should control value subscription. Real life implementation should register remote callback or listener for value update events. Simulation example just fires events in local thread on specified timer trigger.

Method
`public Request getRequest();`
Explanation
Only method that differs from SimpleMonitor interface. It returns request object which identifies all responses from this monitor. All other methods are reflected directly on DAL API.

Best illustration that MonitorProxy should do is simply to observe simulation implementation and comment it.

```
package com.cosylab.datatypes.simulation;

import com.cosylab.datatypes.DataExchangeException;
import com.cosylab.datatypes.Request;
import com.cosylab.datatypes.ResponseListener;
import com.cosylab.datatypes.impl.RequestImpl;
import com.cosylab.datatypes.impl.ResponseImpl;
import com.cosylab.datatypes.proxy.MonitorProxy;

import java.util.TimerTask;

/**
 * Simulation implementation of MonitorProxy.
 *
 * @author Igor Kriznar (igor.kriznarATcosylab.com)
 */
public class MonitorProxyImpl extends RequestImpl implements MonitorProxy,
      Runnable
{
      private PropertyProxyImpl<?> proxy;
      private long timerTrigger = 1000;
      private boolean heartbeat = true;
      private TimerTask task;

      /**
       * Creates new instance.
       *
       * @param proxy parent proxy object
       * @param l listener for notifications
       */
      public MonitorProxyImpl(PropertyProxyImpl proxy, ResponseListener l)
      {
            super(proxy, l);
            this.proxy = proxy;
            resetTimer();
      }
```

As already shown in example of PropertyProxy implementation, MonitorProxyImpl is created with reference to proxy in which was created and responde to listener, who will receive value updates.

Note that MonitorProxyImpl is extended from RequestImpl, which is default support implementation.

```
	public Request getRequest()
	{
		return this;
	}
	public long getTimerTrigger() throws DataExchangeException
	{
		return timerTrigger;
	}
	public void setTimerTrigger(long trigger)
		throws DataExchangeException, UnsupportedOperationException
	{
		timerTrigger = trigger;
		resetTimer();
	}
```

Timer trigger is the rate at which regulate value updates event that are sent to listener but only if monitor is in heartbeat mode.

```
	public void setHeartbeat(boolean heartbeat)
		throws DataExchangeException, UnsupportedOperationException
	{
		this.heartbeat = heartbeat;
		resetTimer();
	}
	public boolean isHeartbeat()
	{
		return heartbeat;
	}
	public long getDefaultTimerTrigger() throws DataExchangeException
	{
		return 1000;
	}
	public boolean isDefault()
	{
		return true;
	}
```

Simulation monitor always returns true for default since it is generating events by itself.

```
	private void fireValueEvent()
	{
		ResponseImpl r = new ResponseImpl(proxy, this, proxy.getValueSync(),
				"value", true, null, proxy.getCondition(), false);
		addResponse(r);
	}
```

This method dispatches events to the listener.

```
	public void fireValueChange()
	{
		if (!heartbeat) {
			fireValueEvent();
		}
	}
```

This method is called from proxy when value is changed and dispatches event but only if monitor is not in heartbeat mode.

```
	public void run()
	{
		fireValueEvent();
	}
	private synchronized void resetTimer()
	{
		if (task != null) {
			task.cancel();
		}
		if (heartbeat) {
			task = SimulatorPlug.getInstance().schedule(this, timerTrigger);
		}
	}
}
```

SimulatorPlug implementation has scheduler, which calls run() method on this monitor proxy at specified rate. In realistic case this should be done by communication layer. Scheduler is implemented with Java Times class and thread pool from Concurrent library.

2.3.5 DirectoryProxy Methods

Best practice is to implement this interface is the same class as PropertyProxy. Especially if communication layer provides a way of introspection with which it is possible to determine information about property. If the information is not possible to get from property it must be obtained in different ways. As to be read from some central directory service. If such service is not available than it could be read from local Map container (loaded with data from some configuration file or hard coded in code). The same DeviceProxy interface is also used for device implementation so all methods, that are marked as device relevant, should be ignored (throw unsupported exception).

Implementation of simulation is a good example for those, that do not have introspection in communication layer. In a simulation all characteristics are stored locally in a the Map. Simulation code example contains only relevant parts.

```
package com.cosylab.datatypes.simulation;

public class PropertyProxyImpl<T> extends AbstractProxyImpl implements PropertyProxy<T>,
SyncPropertyProxy<T>, DirectoryProxy {

	protected Map<String, Object> characteristics = new HashMap<String, Object>();

	/**
	 * Creates new instance.
	 * @param name
	 */
	public PropertyProxyImpl(String name) {
		super(name);
		characteristics.put(NumericPropertyCharacteristics.C_DESCRIPTION,
                                                        "Simulated Property");
		characteristics.put(NumericPropertyCharacteristics.C_DISPLAY_NAME, name);
		characteristics.put(NumericPropertyCharacteristics.C_POSITION, new
                                                            Double(0));
		characteristics.put(NumericPropertyCharacteristics.C_PROPERTY_TYPE,
                                                            "property");
		characteristics.put(NumericPropertyCharacteristics.C_RESOLUTION, 0xFFFF);
		characteristics.put(NumericPropertyCharacteristics.C_SCALE_TYPE, "linear");
		characteristics.put(NumericPropertyCharacteristics.C_UNITS, "amper");
	}
```

When created, the characteristic map is filled with default values.

```
	public String[] getCharacteristicNames() throws DataExchangeException {
		return characteristics.keySet().toArray(new
					String[characteristics.keySet().size()]);
	}
	public Object getCharacteristic(String characteristicName) throws
					DataExchangeException {
		return characteristics.get(characteristicName);
	}
	public Request getCharacteristics(String[] characteristics, ResponseListener
					callback) throws DataExchangeException {
		RequestImpl r= new RequestImpl(this,callback);
		for (int i = 0; i < characteristics.length; i++) {
			r.addResponse(new
				ResponseImpl(this,r
					,this.characteristics.get(characteristics[i])
					,characteristics[i],true,null,condition,true));
		}
		return r;
	}
```

All other methods defined by DirectoryProxy are not property relevant.

2.4 Device Implementation

Similar to property implementation it is the implementation of a device necessary to implement DeviceProxy and DirectoryProxy. Same suggestion holds here: if it is possible then both interfaces should be implemented by the same class.

Device can be understand as a holder for properties and commands. The way that device proxy is designed it is possible to model communication layer with devices even if there is no direct representation in communication layer. Basics for this could be naming structure, where same pattern of properties is repeated over several “logical” devices. Also commands can be modeled with properties: command is simply operation where certain property is set to predefined value.

2.4.1 DeviceProxy and DirectoryProxy Methods

DeviceProxy covers commands and provides child properties. DirectoryProxy provides information which properties and commands are available on the system. If such information is provided from communication layer than implementation is easy.

Method

```
	public CommandProxy getCommand(String name) throws RemoteException;
```

Explanation

Returns proxy class for particular command. Knowledge which commands are actually available is in DirectoryProxy.

Code sample from Simulation

```
	protected Map<String,CommandProxy> commands =  new
						HashMap<String,CommandProxy>();
```

DeviceProxyImpl initialization part.

```
	public CommandProxy getCommand(String name) throws RemoteException {
		return commands.get(name);
	}
```

DeviceProxy method implementation.

```
	public String[] getCommandNames() throws DataExchangeException {
		return commands.keySet().toArray(new String[commands.size()]);
	}
```

Method

DirectoryProxy method implementation. Somebody needs to fill commands array with implementations of CommandProxy interface. Simulation by itself can not know, which commands should be supported.

Method

```
public PropertyProxy getPropertyProxy(String name) throws RemoteException;
public DirectoryProxy getDirectoryProxy(String name) throws RemoteException;
```

Explanation

Both methods return proxies which are then use to create property with particular name. This properties are then used as child nodes of parent device.

Code sample from Simulation

```
protected SimulatorPlug plug;
protected Map<String,DirectoryProxy> directoryProxies;
protected Map<String,PropertyProxy> propertyProxies;
protected Map<String,CommandProxy> commands =  new
                                              HashMap<String,CommandProxy>();
protected Map<String,Class<? extends SimpleProperty>> propertyTypes= new
                            HashMap<String,Class<? extends SimpleProperty>>();
```

DeviceProxyImpl initialization part.

```
public String[] getPropertyNames() {
      return propertyTypes.keySet().toArray(new String[propertyTypes.size()]);
}
public Class<? extends SimpleProperty> getPropertyType(String propertyName) {
      return propertyTypes.get(propertyName);
}
```

DirectoryProxy methods implementation. Simulation by itself can not know, which properties should be supported, somehow information about available properties and their type must be provided.

```
public PropertyProxy getPropertyProxy(String name) throws RemoteException {
      if (propertyProxies==null) {
            propertyProxies= new HashMap<String,PropertyProxy>(3);
      }

      PropertyProxy p= propertyProxies.get(name);

      if (p!=null) {
            return p;
      }

      p = plug.getPropertyProxy(
            this.name+'/'+name,
            SimulatorUtilities.
                  GetPropertyProxyImplementationClass(
                                                      getPropertyType(name)));
      propertyProxies.put(name,p);
      return p;
}
public DirectoryProxy getDirectoryProxy(String name) throws RemoteException {
      if (directoryProxies==null) {
            directoryProxies= new HashMap<String,DirectoryProxy>(3);
      }

      DirectoryProxy p= directoryProxies.get(name);

      if (p!=null) {
            return p;
      }
```

Method

`p = plug.getDirectoryProxy(this.name+'/'+name);`
`directoryProxies.put(name,p);`
`return p;`
`}`

DirectoryProxy implementation returns property proxy and directory proxy. If there is a notion of remote device object in communication library, then implementation may return property proxies initialized by remote property objects, which are obtained from the device. If communication library consist of properties only and device is logically modeled, that similar technique may be used as in simulation implementation. Here properties are obtained from the plug.

2.4.2 CommandProxy

Command proxy is a wrapper for remote command or action. Communication plug may support such object directly or supports remote methods, which can be called from within this proxy, or it does not support commands as such but they can be modeled by invoking certain value on remote data channel. All this can be enveloped with CommandProxy.

API of CommandProxy is almost identical to Command DAL API. The only difference is that command proxy contains synchronous execution method beside method for asynchronous execution.

CommandProxy must support synchronous execution. If command also supports asynchronous execution, then the method

```
public boolean isAsyncrhonous();
```

must return true and the method

```
public Request execute(ResponseListener callback, Object... parameters)
                                                        throws RemoteException;
```

must be implemented.

2.5 AbstractPlug Implementation

The implementation of plug object is done by extension of AbstractPlug. This is because AbstractPlug is the key object for managing PropertyProxy, DeviceProxy and DirectoryProxy life cycle. If communication layer does not support device or properties are obtained directly from devices, than corresponding unused methods may be committed.

Particular implementation of AbstractPlug must be created as a singleton instance. It is recommended that method name **`public static XPlug getInstance()`** is used. This means that only one instance of plug implementation is alive in whole JVM. AbstractPlug also ensures that for each unique name only one instance of PropertyProxy, DeviceProxy or DirectoryProxy is created. For this reason AbstractPlug stores all alive instances of proxies in cache.

Method
`protected AbstractPlug() {...}`
Explanation
Constructor of AbstractPlug is under protected access. Also class extending AbstractPlug must not make constructor visible. Best approach is to make constructor private and obtain

Method

plug instance trough singleton programming pattern as it is done for simulation plug.

Code sample from Simulation

```
package com.cosylab.datatypes.simulation;

public class SimulatorPlug extends AbstractPlug {

	/**
	 * Singleton.
	 */
	private static SimulatorPlug instance;

	public static final SimulatorPlug getInstance() {
		if (instance==null) {
			instance= new SimulatorPlug();
		}
		return instance;
	}

	private SimulatorPlug(){
		super();
	}
```

Method

```
	public abstract String getPlugType();
```

Explanation

Type of plug or communication protocol. For simulation is “Simulator” for EPICS protocol should be “EPICS”, etc.

Code sample from Simulation

```
	public String getPlugType() {
		return "Simulator";
	}
```

Method

```
	protected abstract <T extends PropertyProxy> T createNewPropertyProxy(String
                                    uniqueName, Class<T> type) throws RemoteException;
```

Explanation

Implementation of plug must create new PropertyProxy implementation instance with this method. Class of this instance is provided as method parameter. If it is not possible to create proxy with this name or this implementation class, than exception must be thrown. If type parameter is PropertyProxy.class than plug must decide, which proxy implementation will use. If it is possible to determine type of remote object from communication layer, than decision should be taken regarding remote type.
Method must return as fast as possible. Even if that means that proxy is not connected by the time it is returned.
If the same implementation class is used for PropertyProxy and DirectoryProxy, created proxy must be manually put to directory cache in order not to mix up life cycle procedure: **`putDirectoryProxyToCache(newPropertyAndDirectoryProxy);`**.

Code sample from Simulation

```
	protected <TT extends PropertyProxy> TT createNewPropertyProxy(String uniqueName,
                                        Class<TT> type) throws RemoteException {
		try {
```

Method

```
                if (type==PropertyProxy.class) {
                      PropertyProxyImpl p= new PropertyProxyImpl(uniqueName);
                      putDirectoryProxyToCache(p);
                      return type.cast(p);
                }
                if (!PropertyProxyImpl.class.isAssignableFrom(type)) {
                      throw
                        new IllegalArgumentException(
                                    "Simulator plug can not instantiate class "
                                                            +type.getName());
                }
                PropertyProxyImpl p= (PropertyProxyImpl)type.getConstructor(
                                          String.class).newInstance(uniqueName);
                // adding to directory cache as well
                putDirectoryProxyToCache(p);
                return type.cast(p);
          } catch (Exception e) {
                throw new RemoteException(
                           this,
                           "Failed to instantiate simulation proxy '"
                                +uniqueName+"' for type '"+type.getName()+"'.",e);
          }
    }
          return name;
    }
```

PropertyProxyImpl implements DirectoryProxy as well, so when PropertyProxyImpl instance is created, it is manually added to DirectoryProxy cache. To PropertyProxy cache it is added automatically.

Method

```
    protected abstract DirectoryProxy createNewDirectoryProxy(String uniqueName);
```

Explanation

Implementation of plug must create new DirectoryProxy implementation instance with this method. If same implementation class is used for PropertyProxy and DirectoryProxy, than this method must throw an exception.

Code sample from Simulation

```
    protected DirectoryProxy createNewDirectoryProxy(String uniqueName) {
          throw new RuntimeException(
          "Error in factory implementation, PropertyProxy must be created first.");
    }
```

Here exception is thrown, because when implementation of PropertyProxy is created it is added to the cache of DirectoryProxies as well. So all subsequent calls for directory proxy with the same name as property proxy will return instance from cache.

Method

```
    protected abstract <T extends DeviceProxy> T createNewDeviceProxy(
                        String uniqueName, Class<T> type) throws ConnectionException;
```

Explanation

Implementation of plug must create new DeviceProxy implementation instance with this method. Class of this instance is provided as method parameter. If it is not possible to create proxy with this name or this implementation class, than exception must be thrown. If type parameter is DeviceProxy.class than plug must decide, which proxy implementation will use. If it is possible to determine type of remote object from communication layer, than decision should be taken regarding remote type.

Method

Method must return as fast as possible. Even if that means that proxy is not connected by the time it is returned.
If the same implementation class is used for DeviceProxy and DirectoryProxy, that created proxy must be manually put to directory cache in order not to mix up life cycle procedure: **`putDirectoryProxyToCache(newPropertyAndDirectoryProxy);`**.

Code sample from Simulation

```
      protected <T extends DeviceProxy> T createNewDeviceProxy(String uniqueName,
              Class<T> type) throws ConnectionException {
            try {
                  if (type==DeviceProxy.class) {
                        DeviceProxyImpl p= new DeviceProxyImpl(uniqueName);
                        putDirectoryProxyToCache(p);
                        return type.cast(p);
                  }
                  if (!DeviceProxyImpl.class.isAssignableFrom(type)) {
                        throw new IllegalArgumentException(
                              "Simulator plug can not instantiate class "+
                                                            type.getName());
                  }
                  DeviceProxyImpl p= (DeviceProxyImpl)type.getConstructor(
                                          String.class).newInstance(uniqueName);
                  // adding to directory cache as well
                  putDirectoryProxyToCache(p);
                  return type.cast(p);
            } catch (Exception e) {
                  throw new ConnectionException(this,
                        "Failed to instantiate simulation proxy '"+uniqueName+
                                          "' for type '"+type.getName()+"'.",e);
            }
      }
```

Exception is thrown, because when implementation of PropertyProxy is created it is added to the cache of DirectoryProxies as well. So all subsequent calls for directory proxy with same name as property proxy will return instance from cache.

Method

```
public String getUniqueName();
```

Explanation

Unique name is a name of remote property or data channel.

Code sample from Simulation

```
      public String getUniqueName() {
            return name;
      }
```

where name is obtained from a constructor.

Method

```
public String getUniqueName();
```

Explanation

Unique name is a name of remote property or data channel.

Code sample from Simulation

```
      public String getUniqueName() {
            return name;
      }
```

Method
where name is obtained from a constructor.

Method
`public String getUniqueName();`
Explanation
Unique name is a name of remote property or data channel.
Code sample from Simulation

```
public String getUniqueName() {
	return name;
}
```

where name is obtained from a constructor.

2.6 PropertyFactory Implementation

The PropertyFactory instance freates new properties. How property is obtained from factory does not depend on underlaying system, if it has devices (like ACS or Tango) or flat structure of properties (EPICS). In both ways properties can be obtained from a property factory and plug object takes care of all necessary connections.

There could be several different instances of same property factory class, but they must all use one instance of particular AbstractPlug implementation, which is done using singleton programming pattern as described in AbstractPlug section.

PropertyFactory implementation must provide following methods.

Method

```
public LinkPolicy getLinkPolicy();
public AbstractApplicationContext getApplicationContext();
public void initialize(AbstractApplicationContext ctx, LinkPolicy policy);
```

Explanation
LinkPolicy and ApplicationContext are defined when factory is created. LinkPolicy determine what is default connection behavior when properties are connected.
Code sample from Simulation

```
public LinkPolicy getLinkPolicy() {
	return linkPolicy;
}
public AbstractApplicationContext getApplicationContext() {
	return aac;
}
public void initialize(AbstractApplicationContext ctx, LinkPolicy policy) {
	this.aac = ctx;
	this.linkPolicy = policy;
}
```

Method

```
public PropertyFamily getPropertyFamily();
```

Explanation
Returns property family, which stores all by factory created properties.

Method

Code sample from Simulation

```
	private PropertyFamilyImpl propertyFamily;

	public PropertyFactoryImpl(){
		propertyFamily = new PropertyFamilyImpl(this);
	}
	public PropertyFamily getPropertyFamily() {
		return propertyFamily;
	}
```

Method

```
	public SimpleProperty getProperty(String uniqueName)
                                    throws InstantiationException, RemoteException;
	public <P extends SimpleProperty> P getProperty(String uniqueName, Class<P> type,
		ConnectionListener l) throws InstantiationException, RemoteException;
```

Explanation

First method allows factory and plug to decide, what kind of property will be created for unique remote name. Second method provides desired property type as parameter. Parameter must be a Class object of some DAL property interface.
Factory must perform following steps:

1. If a property already exists in default family it is returned from the family.
2. If a property type is not defined, then factory or plug can decide what type of property will be created.
3. Factory decides which common plug implementation of property to use. There is a helper method that does the job: **`PropertyUtilities.getImplementationClass(type)`**.
4. Factory makes instance of property implementation.
5. Factory decides which implementation of PropertyProxy to use. Simulator does this as follows: **`SimulatorUtilities.getProxyImplementationClass(property)`**.
6. Factory creates property proxy: **`SimulatorPlug.getInstance().getPropertyProxy(uniqueName, proxyType);`**
7. Factory initializes property implementation with proxy.
8. Property is returned.

Code sample from Simulation

```
	public SimpleProperty getProperty(String uniqueName) throws
InstantiationException, RemoteException {
		return getProperty(uniqueName, SimpleProperty.class, null);
	}
	public <P extends SimpleProperty> P getProperty(String uniqueName, Class<P> type,
		ConnectionListener l) throws InstantiationException, RemoteException
	{
		if (propertyFamily.contains(uniqueName)) {
			return type.cast(propertyFamily.get(uniqueName));
		}
		try{
			// Creates property implementation
			Class impClass = PropertyUtilities.getImplementationClass(type);
			DynamicValuePropertyImpl property =
				(DynamicValuePropertyImpl)impClass.
					getConstructor(String.class,PropertyFamily.class).
						newInstance(uniqueName, propertyFamily);
			// creates proxy implementation
			PropertyProxy proxy= SimulatorPlug.getInstance().
				getPropertyProxy(uniqueName,
				SimulatorUtilities.getProxyImplementationClass(property));

			DirectoryProxy dir= SimulatorPlug.getInstance().
```

Method

```
                                                getDirectoryProxy(uniqueName);
                property.initialize(proxy,dir);
                DirectoryProxy dir= SimulatorPlug.getInstance().
                                                getDirectoryProxy(uniqueName);
                property.initialize(proxy,dir);
                propertyFamily.add(property);
                if (l!=null) {
                      property.addLinkListener(l);
                }
                return type.cast(property);
          } catch (Exception e){
                throw new RemoteException(this,
                                        "Failed to instantiate simulation.",e);
          }
    }
```

2.7 DeviceFactory Implementation

The DeviceFactory instance created new devices. How device is obtained from factory does not depend on underlaying system, if it has devices (like ACS or Tango) or flat structure of properties (EPICS). In both ways device can be obtained from device factory and plug object takes care of all necessary connections or logical modeling of device (eg. from naming hierarchy). In general property factory it is more important to be implemented than directory factory, since properties are basic objects through which remote value is accessed.

There could be several different instances of same device factory class, but they must all use one instance of particular AbstractPlug implementation, which is done by use of singleton programming pattern as described in AbstractPlug section.

DeviceFactory implementation must provide following methods.

Method

```
    public LinkPolicy getLinkPolicy();
    public AbstractApplicationContext getApplicationContext();
    public void initialize(AbstractApplicationContext ctx, LinkPolicy policy);
```

Explanation

LinkPolicy and ApplicationContext are defined when factory is created. LinkPolicy determine what is default connection behavior when devices are connected.

Code sample from Simulation

```
    public LinkPolicy getLinkPolicy() {
          return linkPolicy;
    }
    public AbstractApplicationContext getApplicationContext() {
          return aac;
    }
    public void initialize(AbstractApplicationContext ctx, LinkPolicy policy) {
          this.aac = ctx;
          this.linkPolicy = policy;
    }
```

Method

```
    public DeviceFamily getDeviceFamily();
```

Explanation

Method

The Device family stores all by factory created devices.

Code sample from Simulation

```
	private DeviceFamilyImpl deviceFamily;

	public DeviceFactoryImpl(){
		deviceFamily = new DeviceFamilyImpl(this);
	}
	public DeviceFamily getDeviceFamily() {
		return deviceFamily;
	}
```

Method

```
	public AbstractDevice getDevice(String uniqueName)
                                    throws InstantiationException, RemoteException;
	public <P extends AbstractDevice> P getDevice(String uniqueName, Class<P> type,
		ConnectionListener l) throws InstantiationException, RemoteException;
```

Explanation

The first method allows factory and plug to decide, what kind of device will be created for unique remote name. The second method provides desired device type as parameter. Parameter must be Class object of some DAL device interface.
Factory must perform following steps:

9. If device already exists in default family it is returned from the family.
10. If a device type is not defined, then factory or plug can decide what kind of device will be created.
11. Factory decides which common plug implementation of device to use.
12. Factory makes instance of device implementation.
13. Factory decides which implementation of DeviceProxy to use. Simulator does this as follows: **`SimulatorUtilities.getProxyImplementationClass(device)`**.
14. Factory creates device proxy:
    `SimulatorPlug.getInstance().getDeviceProxy(uniqueName, proxyType);`
15. Factory initializes device implementation with proxy.
16. Device is returned.

Code sample from Simulation

```
	public AbstractDevice getDevice(String uniqueName) throws InstantiationException,
RemoteException {
		return getDevice(uniqueName, AbstractDevice.class, null);
	}
	public <P extends AbstractDevice> P getDevice(String uniqueName, Class<P> type,
		ConnectionListener l) throws InstantiationException, RemoteException
	{
		if (deviceFamily.contains(uniqueName)) {
			return type.cast(deviceFamily.get(uniqueName));
		}
		try{
			// Creates device implementation
			// TODO: missing proper implementation
			//Class impClass = DeviceUtilities.getImplementationClass(type);
			//AbstractDeviceImpl device=(AbstractDeviceImpl)impClass.
			//	getConstructor(
			//	String.class, DeviceFamily.class).newInstance(uniqueName,
			//	propertyFamily);
			AbstractDeviceImpl device = new AbstractDeviceImpl(uniqueName);
			connect(uniqueName, type, device);
			family.add(device);
			if (l != null) {
				device.addLinkListener(l);
```

Method

```
                }
                return type.cast(device);
            } catch (Exception e){
                throw new RemoteException(this,
                                          "Failed to instantiate simulation.",e);
            }
        }
```

3. Loading Different Plug Implementations

PropertyFactory and PropertyFactory instances are obtained from DefaultPropertyFactoryService and DefaultDeviceFactoryService. These services are default implementations of PropertyFactoryService and DeviceFactoryService. This section describes behavior of default implementations.

They decide which factory to use in two ways:

1. Plug type is provided as parameter, code snippet is below:

```
public interface DeviceFactoryService {
      public DeviceFactory getDeviceFactory(AbstractApplicationContext ctx,
                                            LinkPolicy linkPolicy, String plugName);
}
public interface PropertyFactoryService {
      public PropertyFactory getPropertyFactory(AbstractApplicationContext ctx,
                                            LinkPolicy linkPolicy, String plugName);
}
```

   In this case default service implementation tries to load defined plug. Which class to load for provided plug name should be provided trough configuration as it is discussed after the next point.

2. Default plug type is obtained from configuration of ApplicationContext.

Configuration parameters in ApplicationContext can be hard coded or loaded from the some configuration file or provided as command line parameter for system properties. At the moment there is no default way to define them. But in the future it may be convenience class, which will provide integration with underlaying application framework (such as RCP from Eclipse).

ApplicationContext snippet:

```
public interface AbstractApplicationContext extends LifecycleReporter,
            Identifiable {
      public Properties getConfiguration();
}
```

If the plug type for factory is not provided as parameter to factory service then default factory service uses two distinct configuration keyword methods to determine which default implementation factory class should be used.

1. Keywords directly defining implementation classes:

```
PropertyFactoryService.DEFAULT_FACTORY_IMPL ==
                                    "PropertyFactoryService.default_factory_impl"
DeviceFactoryService.DEFAULT_FACTORY_IMPL ==
                                     "DeviceFactoryService.default_factory_impl"
```

   Factory service first checks configuration for these keywords, if default implementation is requested. These keywords must store valid class name of factory implementation.

2. As second resort factory service checks plug definitions.

```
Plugs.PLUGS == "dal.plugs"
```

This keyword value must contain comma separated list of available plug names. Plug name must match the name returned by corresponding plug implementation. For example:

```
Properties conf= new Properties();
conf.setProperty("dal.plugs","Simulator,EPICS,TANGO");
```

Each plug name in list property factory and device factory must be defined by using corresponding prefix

```
Plugs.PLUG_DEVICE_FACTORY_CLASS == "dal.devicefactory."
Plugs.PLUG_PROPERTY_FACTORY_CLASS == "dal.propertyfactory."
```

and plug name. Continuation of previous example follows:

```
conf.setProperty("dal.devicefactory.Simulator",
                         "com.cosylab.datatypes.simulation.DeviceFactoryImpl");
conf.setProperty("dal.propertyfactory.Simulator",
                         "com.cosylab.datatypes.simulation.PropertyFactoryImpl");

conf.setProperty("dal.devicefactory.EPICS","org.epics.dal.DeviceFactoryImpl");
conf.setProperty("dal.propertyfactory.EPICS","org.epics.dal.PropertyFactoryImpl");

conf.setProperty("dal.devicefactory.TANGO","org.tango.dal.DeviceFactoryImpl");
conf.setProperty("dal.propertyfactory.TANGO","org.tango.dal.PropertyFactoryImpl");
```

Factory service then loads appropriate class when factory for particular plug is requested. If plug is not provided as parameter to factory service, default plug is chosen. ApplicationContext configuration below defines which plug is default:

```
Plugs.PLUGS_DEFAULT == "dal.plugs.default"
```

For example:

```
conf.setProperty("dal.plugs.default","Simulator");
```

For handling configuration **`Plugs`** class can be used. It also provides conveniende methods for extracting plug information from configuration.

4. Directory Service And DAL

Directory service is authority that provides access to metadata: it can describe the system without actually making connection to the system. In this sense it is out of the scope of DAL. Directory is not directly part of DAL, it is rather service parallel to DAL. DAL may use directory service to retrieve additional information or to store some cached values, but this is DAL implementation dependant. DAL plug may also fill directory with information about itself (version, supported features, etc.), this part has not been defined yet.

Important consideration for any implementation of directory service is not to use DAL directly. This my create close loop if some DAL implementation uses directory service.

By designed DAL does not need direct reference to directory service, still plug implementation has to provide some introspection functionality by DirectoryProxy interface. Implementation of DirectoryProxy is completely arbitrary, some communication protocol may provide this functionality directly. If particular communication protocol does not provide this functionality, DirectoryProxy implementation may ask directory service or some other meta data or introspection authority.

There is distinction between directory service and directory proxy. Information that must be provided from a DirectoryProxy in order for DAL to function:

- names of avaliable characteristic for particular data channel,
- value of particular channel's characteristic,
- name of avaliable command names for particular data channel,
- names of avaliable properties (data channels) for particular device,
- data type of particular property (data channel)

From DAL it is expected to make connection and return data for particular remote object if remote name is provided. Functionality of DirectoryProxy provides DAL with additional information required to execute remote request.

Information what is availaible in particular system is in domain of directory service. Without defining API a folowing functionality can be identified of such service:

- All what has been already defined for DirectoryProxy.
- List of avaliable data channels.
- List of avaliable devices.
- List of avaliable DAL protocol implementations (plugs).
- Protocol specific detailed information (protocol version, protocol dialect, supported functionality in underlaying protocol implementation different from standard implementation).
- Implementation and data type of channels and devices
- Search capability for channels and devices.
- Detailed information about special protocol capabilities: is “BLUE BEAM” supported by event system, etc.

DAL itself does not specify how directory service should be implemented or how API should be used. The directory service is a service parallel to DAL and they both are used from application layer.

JNDI is ideal API for Java directory service, that would work well with DAL concept. JNDI is Java interface for naming and directory service with some interesting default implementations, like LDAP.

DAL prototype implementation define how JNDI directory service is used. Following code initializes JNDI staring point in DAL, an initial Context instance:

```
Properties p = System.getProperties();
URL url = ClassLoader.getSystemResource("jndi.properties");
FileInputStream fis = new FileInputStream(url.getFile());
p.load(fis);

DirContext initialContext = (DirContext) NamingManager.getInitialContext(p);
```

In this case it is assumed that the file jndi.properties is included in classpath (by adding dal jar to classpath for example) and this file includes declaration of initial Context implementation Class. The following line must be included in jndi.properties file:

```
java.naming.factory.initial=org.epics.css.dal.directory.InitialContextFactoryImpl
```

DAL initial context loader takes care that each time same instance of initial context is returned in the same JVM. There is also convenience implementation, which performs

the same initalization as above:

```
DirContext initialContext= DirectoryUtilities.getInitialContext();
```

The initial JNDI context can be queried for meta information about channels. Queries are submitted in form of URI strings on URIName objects. The URI string has a following notation:

```
DAL-<PLUG NAME>://<AUTHORITY>/<REMOTE NAME>
```

Where:

- DAL- is prefix, that defines particular URI to be part of DAL namespace.
- <PLUG NAME> is a name of protocol, like Simulation of EPICS.
- <AUTHORITY> is URI authority, like IP of computer hosting remote object. Particular protocol may require that authority part can have arbitrary value, only some default value or may be omitted.
- <REMOTE NAME> is name of remote object (channel, device or their components). It may be further fragmented in hierarchy by / characters or may be flat namespace. Can be hierarchical name as described in [5].

Proposed naming scheme and proper implementation of JNDI interface allows to send queries to LDAP implementation of JNDI as suggested in [5]. For example PV named 'fref' with hierarchical path `"fred.linac.sns.epics"` and LDAP name `"cn=fred,dc=linac,dc=sns,dc=epics"` would have DAL URI like this: `"DAL-EPICS:///epics/sns/linac/fred"`.

Example based on Simulator, which performs lookup in two ways: one with URI string, second with URIName instance is written bellow:

```
String uri= "DAL-Simulator:///PS1:current";
URIName uname= new URIName("DAL-Simulator",null,”PS1:current”,null);
Object o1 = initialContext.lookup(uri);
Object o2 = initialContext.lookup(uname);
if (o1==o2) OK;
```

Note, that DAL at this point does not specify what object should be binded in directory under given URI name. Neither does DAL use this object. This can be defined in the future by the application layer, where such information may be used.

However simulation implementation of DAL uses JNDI directory to store and retrieve characteristics (channel fields) as JNDI attributes. Other DAL implementation may choose similar approach.

The example below is taken from simulation DAL implementation where DirectoryProxy queries JNDI simulation directory for a characteristic value of particular channel. In this case simulation JNDI plays role of cache for simulated values.

```
DirContext initialContext= DirectoryUtilities.getInitialContext();
String uri= "DAL-Simulator:///PS1:current";
String characteristicName= "maxValue";
Attributes attr = ctx.getAttributes(uri);
Object characteristic = attr.get().get();
```

4.1 EPICS Directory Proposal

This section constitutes Cosylab's proposal towards an implementation of an EPICS Name Service. The opinion is based on Cosylab's work on a reliable EPICS Directory Service performed in context of the 'Dependable Distributed Systems' project, [6].

A preliminary list of requirements is given in [5] and extended in DeDiSys Work Package 3.3: Design Documents & First Prototype & Test Report (sec. 4: WP3.3 application scenario). To recap, the requirements are as follows:

- R A1: Wildcard searches. Clients should be able to resolve channels to IOCs using wildcards.
- R A2: Report on number of channels per IOC. The naming service should be able to report how many channels a particular IOC is hosting.
- R A3: Redundant naming services. A hot-standby backup naming service is available, which takes over name resolutions in case of primary's failure.
- R A4: Redundant IOCs with the same PV. The same PV can be hosted on more than one IOC. The naming service is capable of redirecting clients to the IOC most capable of serving a PV (e.g., the active IOC, or the one with least load).
- R A5: Archived data for a PV. Note: this is not the role of a naming service, but more of an archiving or gateway service. Therefore, the request will not be considered at this point in time.
- R A6: Backup/restore locations for a PV.
- R A7: PV meta data. The naming service should store information such as engineer-in-charge, CA security roles, etc.
- R B1: Minimal setup effort. The naming service should configure itself on-the-fly if no configuration is given (e.g., by establishing name—IOC mappings as they are requested using existing name resolution).
- R B2: Lookup performance. The lookup performance should be up to an order of magnitude slower for the first lookup, and as fast as without the naming service for subsequent lookups.
- R B3: Bind performance. The performance of a bind can be up to an order of magnitude slower than the performance of a lookup without the naming service.
- R B4: Scalability. The naming service should not have any particular scalability bounds. It should be capable of handling 1 million records. The algorithms for binding and lookup should be better than O(N2).
- R B5: EPICS V3 backwards compatibility. The naming service shall require no modifications to existing EPICS V3 clients and servers.

In [5], LDAP is proposed as the EPICS Name Service. However, the choice of LDAP has several disadvantages:

1. LDAP is not extensible beyond provisions made by the RFC standard. For example, the following EPICS Name Service requirements are difficult if not impossible to implement with LDAP:
   - R A2 (would require search by attribute and counting, implemented on each client separately)
   - R A3 (depends on particular LDAP server implementation and may be difficult to set up, understand and maintain)
   - R A4 (it is impossible to bind several entries with the same name in LDAP)
   - R B1 (population of LDAP database with EPICS deployment data is

non-trivial)
    - R B2, R B3 (LDAP implementations are relatively slow to other approaches)
    - R B5 (LDAP implementations are difficult to integrate with CA)
2. The on-the-wire protocol of LDAP is fairly complex as it is based on ASN.1's BER encoding of entities defined in LDAP RFC 2251. Thus, it will be difficult to implement LDAP clients or servers if they don't yet exist for a target platform of choice (e.g., VxWorks clients, Java servers, etc.).
3. LDAP is a suitable for naming, not for a meta-data rich directory service required by EPICS, as alrady acknowledged in [5].

In the context of DeDiSys project, an EPICS Directory Service (EPICS DS) is being implemented in the following manner:

- EPICS DS intercepts CA's CA_PROTO_SEARCH requests, which are queries for addresses of IOCs that host a particular PV.
- If EPICS DS already knows the whereabouts of the requested PV, it returns it from local memory, most likely implemented as a hash-table or similarly efficient data structure. This allows for a very high-performance name resolution and binding.
- If the PV is unknown, the EPICS DS broadcasts CA_PROTO_SEARCH request to the destination subnet where IOCs reside. The answer is cached and returned. This allows for minimal setup effort of the DS, as it is populated automatically.
- This approach works also for existing EPICS V3 applications without modifications.
- EPICS DS is designed to be a highly-available service, which will be achieved through replication techniques under consideration in DeDiSys project.

We recommend the following future steps:

1. For Java clients, JNDI [6] APIs are used. To implement meta-data (accessible in form of JNDI attributes) and attribute-based search, 'DirContext' interface should be used ('Context' is insufficient). JNDI schema as proposed in [5] is used to store EPICS PV data.
2. In addition to existing CA_PROTO_SEARCH interface, EPICS DS implements a special-purpose, EPICS-optimized protocol using HTTP GET requests to access its data structures. Also, a JNDI implementation utilizing this HTTP protocol is created.

The product of step 2 will constitute a solid platform for easy access from all platforms, as client-side API is trivial to implement. Also, as a side effect, EPICS DS will be readily available from a web browser. Furthermore, new EPICS DS features will be possible to add without abusing the LDAP protocol further.

I. Appendix – PropertyProxy and DirectoryProxy Interfaces

This is a Proxy that has common methods for property and directory proxy.

```
package com.cosylab.datatypes.proxy;

import com.cosylab.datatypes.context.ConnectionState;
import com.cosylab.datatypes.context.Identifiable;

/**
 * Common interface for all proxies.
 *
 * @author Igor Kriznar (igor.kriznarATcosylab.com)
 *
 */
public interface Proxy extends Identifiable {
	/**
	 * Returns the unique name. This name is used to initiate
	 * connection to remote object and can be regards as remote name.
	 * @return String unique remote name for this property
	 */
	public String getUniqueName();
	/**
	 * Destroys object and releases all remote and local allocated resources.
	 * <p><b>NOTE</b></br>
	 * Only plug which created this proxy can call this method since lifecycle is
	 * controled by the plug.
	 * </p>
	 */
	public void destroy();
	/**
	 * Registers new listener of proxy events.
	 * @param l new listener
	 */
	public void addProxyListener(ProxyListener l);
	/**
	 * Deregisters listener from proxy events.
	 * @param l new listener
	 */
	public void removeProxyListener(ProxyListener l);
	/**
	 * Return connection state enum of the remote object.
	 * @return state of connection to remote object
	 */
	public ConnectionState getConnectionState();
}
```

PropertyProxy interface.

```
package com.cosylab.datatypes.proxy;

import java.util.EnumSet;

import com.cosylab.datatypes.DataExchangeException;
import com.cosylab.datatypes.DynamicValueState;
import com.cosylab.datatypes.RemoteException;
import com.cosylab.datatypes.Request;
import com.cosylab.datatypes.ResponseListener;

/**
 * Interface encapsulate communication to remote property object.
 *
 * @author Blaz Hostnik
 */
public interface PropertyProxy<T> extends Proxy {
	/**
	 * Asynchronously gets remote property value.
	 *
	 * @return a Request, which identifies incoming responses.
	 */
	public Request getValueAsync(ResponseListener callback) throws
	                                                        DataExchangeException;
	/**
	 * Asynchronously sets value on remote property.
	 *
	 * @param value new value.
	 * @return a Request, which identifies incoming responses.
	 */
	public Request setValueAsync(T value, ResponseListener callback) throws
	                                                        DataExchangeException;
	/**
	 * Returns whether the value can be set on remote object presented by this proxy.
	 *
	 * @return <code>true</code> if value can be set.
	 */
	public boolean isSettable();
	/**
	 * Creates new value subscription and returns monitor proxy, which controls the
	 * subscription.
	 * @param callback a listener which will receive subscription events.
	 * @return monitor proxy, which controls the subscription.
	 * @throws RemoteException if operation fails
	 */
	public MonitorProxy createMonitor(ResponseListener callback) throws
	                                                                RemoteException;
	/**
	 * Returns remote condition of this dynamic value representation.
	 * @return remote condition
	 */
	public EnumSet<DynamicValueState> getCondition();
}
```

DirectoryProxy interface, the same interface is used for property and device proxies.

```
package com.cosylab.datatypes.proxy;

import com.cosylab.datatypes.DataExchangeException;
import com.cosylab.datatypes.Request;
import com.cosylab.datatypes.ResponseListener;


/**
 * This interface encapsulate access to introspection part of remote property. Each
 * instance is asociated with
 * particular unique remote name and all in and out names are relative to this name.
 *
 * @author Blaz Hostnik
 */
public interface DirectoryProxy
{
      /**
       * Returns the unique name. This name is used to initiate
       * connection to remote object and can be regardes as remote name.
       * @return String unique remote name for this property
       */
      public String getUniqueName();
      /**
       * Destroys object and releases all remote and local allocated resources.
```

```
     * <p><b>NOTE</b></br>
     * Only plug which created this proxy can call this method since lifecycle is
     * controled by the plug.
     * </p>
     */
    public void destroy();
    /**
     * Returns names of all characteristics for this proxy. Return value will
     * be an array of non-null characteristic names.
     * @return an array of non-null characteristic names
     * @throws DataExchangeException if operation fails
     */
    public String[] getCharacteristicNames() throws DataExchangeException;
    /**
     * Returns available command names, if this proxy represents device.
     * @return all available command names
     * @throws DataExchangeException if remote request fails
     */
    public String[] getCommandNames() throws DataExchangeException;
                                                              DataExchangeException;
    /**
     * Accesses asynchronously the complete map of characteristics for this proxy.
     * This asynchronous request is considered as multiple request where each
     * characteristic name can be treated as separate single request. As consequence,
     * responses to this request are returned independantly. For each charactetistic
     * name one response is returned, where characteristic name is defined by ID tag
     * of response and response value is value of
     * characteristic. Last response is marked as last (isLast() method returns true).
     * @param characteristics list of requested characteristics names.
     * @param callback a callback listener, which will receive all responses
     * @return a Request, which identifies incoming responses.
     * @throws DataExchangeException if operation failes
     */
    public Request getCharacteristics(String[] characteristics, ResponseListener
                                    callback) throws DataExchangeException;
    /**
     * Returns the value of the characteristic. If the characteristic with such name
     * does not exist this method returns <code>null</code>. If the characteristic
     * exists but the value is unknown,
     * <code>CharacteristicContext.UNINITIALIZED</code> is returned.
```

```
         *
         * @param characteristicName the name of the characteristic, may not be
         * <code>null</code> or an empty string
         * @return Object the value of the characteristic or <code>null</code> if unknown
         * @exception DataExchangeException when the query for the characteristic value on
         * the data source fails
         */
        public Object getCharacteristic(String characteristicName) throws
                                                                DataExchangeException;
        /**
         * Returns names of properties if this proxy represents device proxy. Names are
         * relative name, which are valid only in context of this device proxy.
         *
         * @return names of properties in this device proxy
         */
        public String[] getPropertyNames();
        /**
         * Returns data access interface class (extended from SimpleProperty interface)
         * of property in device.
         * Valid only if this directory proxy represents device proxy. Name is relative,
         * valid only in context of this device proxy.
         *
         * @param propertyName name of the property
         * @return data access inerface class
         */
        public Class<? extends SimpleProperty> getPropertyType(String propertyName);
}
```

SyncPropertyProxy interface, property proxy implementation may optionally choose to implement this interface as well.

```
package com.cosylab.datatypes.proxy;

import com.cosylab.datatypes.DataExchangeException;

/**
 * This interface is optional. If property proxy does not implement this
 * interface, than property implementation simulates sincrhounous operations by
 * calling asynchronous version and blocks call untill response is returned.
 *
 * @author Igor Kriznar (igor.kriznarATcosylab.com)
 */
public interface SyncPropertyProxy<T> extends PropertyProxy<T> {
        /**
         * Synchronous get of value.
         * @return returned values
         * @throws DataExchangeException if remote operation fails
         */
        public T getValueSync() throws DataExchangeException;
        /**
         * Synchronous set of new value.
         * @param value new value to set
         * @throws DataExchangeException if remote operation fails
         */
        public void setValueSync(T value) throws DataExchangeException;
}
```

MonitorProxy interface.

```
package com.cosylab.datatypes.proxy;

import com.cosylab.datatypes.Request;
import com.cosylab.datatypes.SimpleMonitor;

/**
```

```
 * MonitorProxy interface is used to control value subscription.
 *
 * @author Igor Kriznar (igor.kriznarATcosylab.com)
 */
public interface MonitorProxy extends SimpleMonitor
{
      /**
       * Returns request object asociated with this value response subscritpion.
       * @return a Request, which identifies incoming responses.
       */
      public Request getRequest();
}
```

```
package com.cosylab.datatypes;

/**
 * Interface <code>SimpleMonitor</code> defines the way in which an observer
 * (realized as JavaBeans listener) can influence the flow of events from the
 * data access event source. Subscription can be changed by each observer separately
 * and it affects that observer only. Event delivery is changed by changing
 * characteristics of the monitor.
 * <p>
 * Timer trigger may operate at the prescribed default frequency, with all other
 * (optional) triggers set to their respective default values. In this case the method
 * <code>isDefault</code> must return <code>true</code>. The default subscription may be
 * handled separately (in a more efficient way) by the underlying implementation.
 * </p>
 * <p>
 * Methods of this interface which generate transient connection objects such as
 * requests and
 * responses, store those objects directly into the latest values of the property to
 * which this monitor belongs.
 * </p>
 */
public interface SimpleMonitor {
      /**
       * Returns the value of the timer trigger for this subscription. The timer trigger
       * is the promise of the implementation of this interface to check for the new
       * value in
       * the underlying data source every <code>long</code> milliseconds. If an event is
       * dispatched in response to the check depends on the <code>heartbeat</code>
       * value. This is a writable characteristic of the subscription.
       * @return long the amount in milliseconds between the checks for new value status
       *                  in the underlying data source layer
       * @exception DataExchangeException if the query for the current trigger value
       * fails
       */
      public long getTimerTrigger() throws DataExchangeException;
      /**
       * Sets the timer trigger for this subscription. The value supplied must be
       * positive or zero. If it is zero, the timer trigger is disabled and no periodic
       * checks will be performed. Of course, other (optional) triggers may still cause
       * an event to be dispatched in this case. This is a writable characteristic of
       * the subscription.
       * @param trigger a positive or zero value in milliseconds between the checks in
       *              the underlying data source layer
       * @exception DataExchangeException if the set of the new trigger fails
       */
      public void setTimerTrigger(long trigger) throws DataExchangeException,
                                                       UnsupportedOperationException;
      /**
       * Sets the heartbeat flag for this subscription. If the heartbeat is turned on,
       * an event is dispatched to the listeners whenever any trigger triggers. If the
       * heartbeat is turned off, the event is dispatched <b>only</b> if the dynamic
       * value or its status has changed. In any case, the
       * <code>latestUpdateTimestamp</code> in <code>Updateable</code> interface will be
       * modified.
       * @param heartbeat <code>true</code> iff the subscription should generate
       *              heartbeat events
       */
      public void setHeartbeat(boolean heartbeat) throws DataExchangeException,
                                                       UnsupportedOperationException;
      /**
       * Returns the value of the heartbeat flag for this subscription.
       *
       * @return boolean <code>true</code> iff the subscription generates heartbeat
       * events
       */
      public boolean isHeartbeat();
      /**
```

```
        * Returns the default timer trigger that is used when the subscription first
        * becomes active. This is a characteristic of the subscription. Its value is
        * determined by the implementation.
        *
        * @return long the default value of the <code>timerTrigger</code> characteristic
        *                  in milliseconds
        * @exception DataExchangeException if the query for the characteristic value
        * fails
        */
       public long getDefaultTimerTrigger() throws DataExchangeException;
       /**
        * Returns <code>true</code> if the subscription has default triggers. The default
        * valuesof all triggers (timer as well as optional) are determined by the
        * implementation of this interface. If the subscription is default, it may be
        * optimized by the underlying layer. It is expected that most of the
        * subscriptions will, during their duration, remain at default triggers. For
        * example, the implementation may choose to create only one link to the
        * underlying (remote) layer for all default subscriptions.
        *
        * @return boolean <code>true</code> if this subscription uses default triggers
        */
       public boolean isDefault();
       /**
        * Destroys this monitor and ends subscription to remote value
        * changes.
        */
       public void destroy();
       /**
        * Returns <code>true</code> if destroy has been called.
        *
        * @return <code>true</code> if destroy has been called
        */
       public boolean isDestroyed();
}
```