Firebird 3: provider-based architecture, plugins and OO approach to API

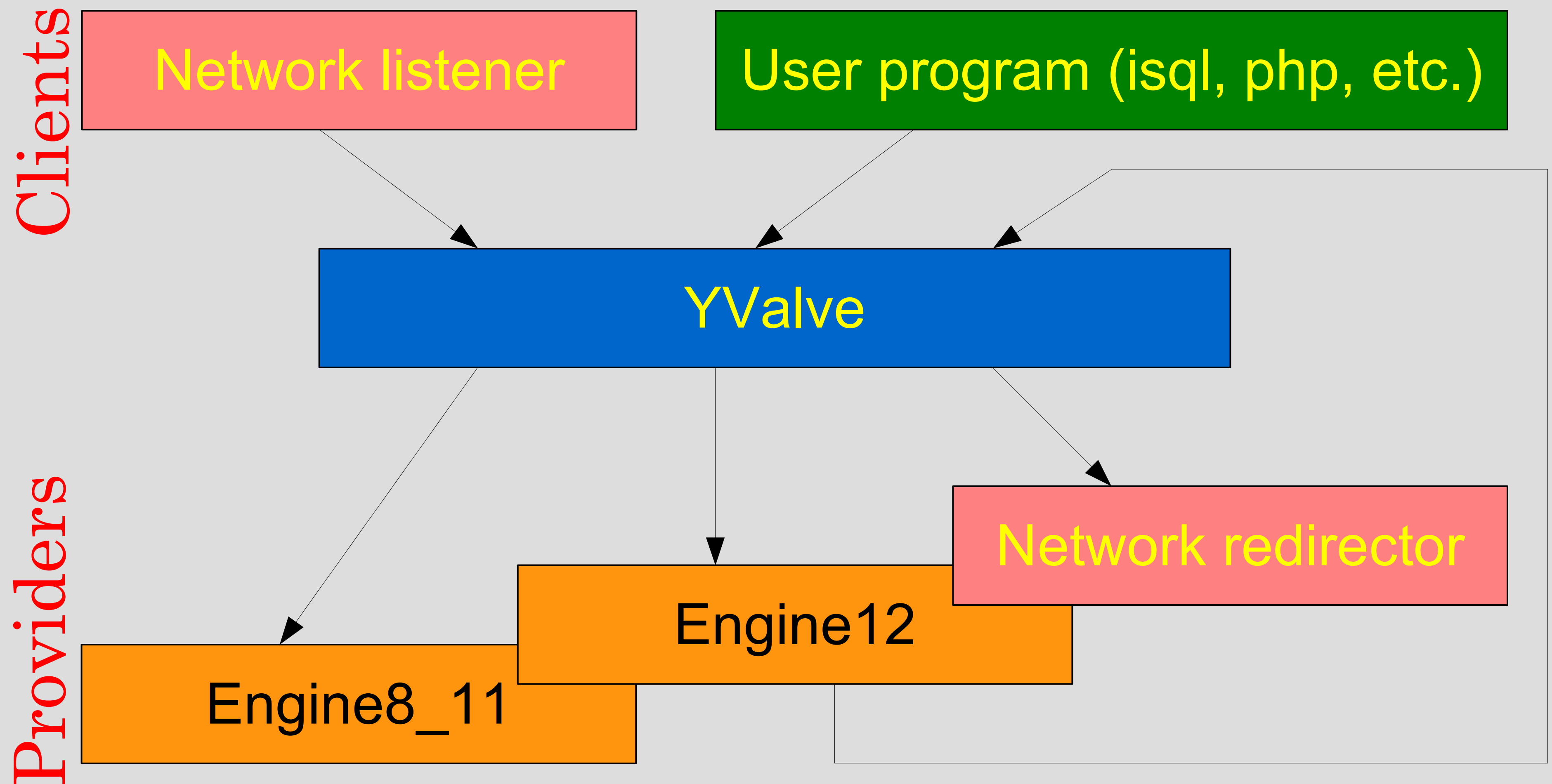
Alex Peshkov

Firebird Foundation
IbPhoenix
2011

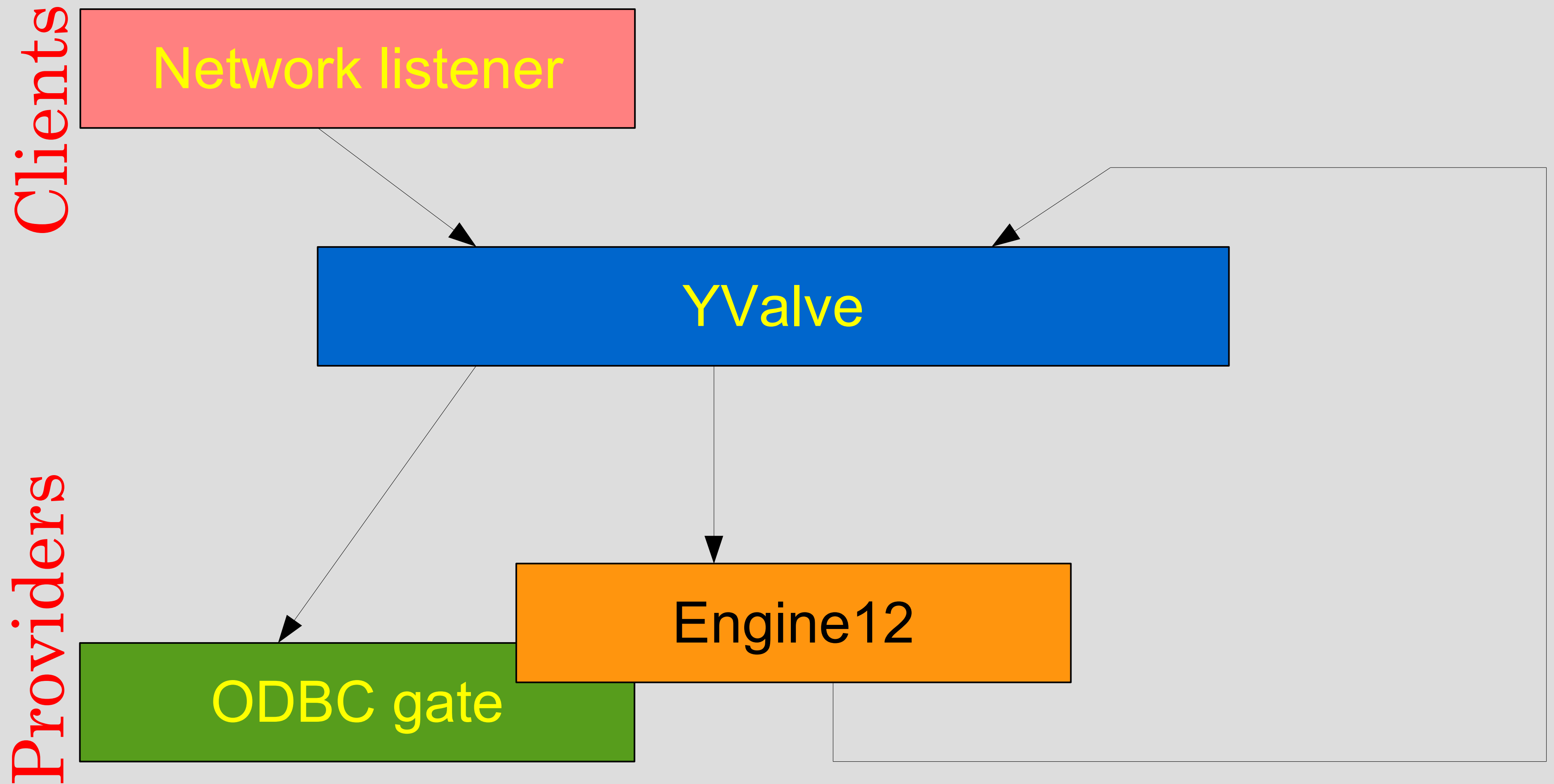
Architectural goals of Firebird3

- Provide more and better ways for users to extend functionality of firebird
- How to do it:
  - Introduce (or may be restore ?) provider-based architecture
  - Introduce (or may be extend ?) plugins

Providers – based architecture (OSRI)



Server with additional provider



Server with additional provider

EXECUTE STATEMENT 'ODBC SPECIFIC OPERATOR'
ON EXTERNAL 'ODBC://odbc_datasource_name'

Engine12:

Attach Database 'ODBC://odbc_datasource_name'

YValve:

Tries known providers, including ODBC gate

ODBC gate:

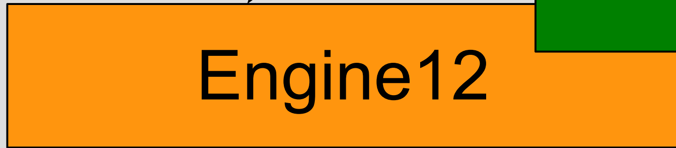
Recognizes 'ODBC://' prefix

Calls appropriate ODBC function to establish connection

Client with additional provider

Clients

Providers



Client with additional provider

Client:

    Attach Database 'CACHE://SRV/dbAlias'

Yvalve:

    Tries known providers, including CACHE

CACHE:

    Recognizes 'CACHE://' prefix
    Can use 'INET://somehost/dbAlias' to access remote data
    Can use embedded connection to access cached data

Plugins

- Provide almost unlimited capabilities of extending firebird with what user needs
  - UDFs and blob filters are also a kind of plugins with specific interface and calling rules
- Require (like UDFs) special care to avoid malicious code, executed in server context
  - all plugins, not described explicitly in configuration file, are loaded from $(root)/plugins, slash in names is not permitted
  - if one provides path information in explicit plugin description – he should care about it himself

Plugins

- May be plugged only into specially prepared points of main firebird code
- Firebird 3 will support the following types of plugins:
  - Trace;
  - External engines;
  - Authentication (server/client) and users' management;
  - Crypt (network and may be database).

Plugins

- Have interface, specific for each predefined point in firebird code
- Have common rules of load/unload and configuration
- Have common interface, controlling that common features

Plugins and providers-base architecture

Provider is invoked from specific point of a code (from yValve)
Providers should be loaded/unloaded to/from firebird process address space

Providers can and should be treated as a special kind of plugins

- this avoids adding special code, duplicating one for plugins

Choosing API

- Functional API
  - Follows existing (isc_attach_database) style
- Object-oriented API
  - Used in a lot of modern software
  - Saves resources when loading plugin
- Backward compatibility at yValve level for providers API

Interfaces

- Firebird API contains 2 types of objects:
  - Interface - C++ class with pure-virtual only functions;

```
class IIntUserField : public IUserField
{
public:
        virtual int FB_CARG get() = 0;
        virtual void FB_CARG set(int newValue) = 0;
};
```

Simplified form, used later

```
class IIntUserField : public IUserField
{
        int get();
        void set(int newValue);
};
```

Interfaces

Firebird API contains 2 types of objects:

- Structure - C struct, containing POD (plain old data) only;

```
struct FbMessage
{
	const unsigned char* blr;
	unsigned char* buffer;
	unsigned int blrLength;
	unsigned int bufferLength;
};
```

Master interface

- Provides access to other interfaces
- Stands separate from the others, cause created (from user POV) not by any other interface, but by API function:
  - IMaster* fb_get_master_interface();
- This is the only one new API function, required to support OO API

Common rules

- All interfaces are derived from IVersioned, IDisposable or IRefCounted (last two are also derived from IVersioned).
- All plugins are derived from IPluginBase (derived from IRefCounted).
- Interfaces guaranteed lifetime:
  - IRefCounted – as long as not released last time,
  - IDisposable – as long as not disposed,
  - non of this (just versioned) – according to lifetime of outer object (which created that interface)

IVersioned – version of interface

Firebird interfaces are not COM interfaces – we support multiple versions of same interface.

Interface version can be upgraded by code, receiving interface from other module.

Interface version is always equal to total number of virtual functions in it.

Use of upgraded interface does not cause any delays when using it's functions.

IVersioned – version of interface

```
class IVersioned {
   int getVersion();
   IPluginModule* getModule();
};
```

IVersioned – version of interface

Upgrade is supported by IVersioned and function:

```
IMaster::upgradeInterface(
        IVersioned* toUpgrade,
        int desiredVersion,
        struct UpgradeInfo* upgradeInfo).

struct UpgradeInfo
{
    void* missingFunctionClass;
    IPluginModule* clientModule;
};
```

IVersioned – version of interface

Samples of missingFunctionClass

```
class NoEntrypoint {
  virtual void FB_CARG noEntry(IStatus* s) {
     s->set(Arg::Gds(isc_wish_list).value());
  }
};

class IgnoreMissing {
  virtual int FB_CARG noEvent() {
    return 1;
  }
}
```

IVersioned – version of interface

Calling functions in upgraded interface

```
class IService : public IRefCounted
{
        void detach(IStatus* status);
        void query(IStatus* status,
             int sendLength, char* sendItems,
             int receiveLength, char* receiveItems,
             int bufferLength, char* buffer);
        void start(IStatus* status,
             int spbLength, char* spb);
};
```

Explicit lifetime control

Required when interface is created at one place and destroyed at another

IDisposable – used in interfaces, not intended to be passed from thread to thread

```
class IDisposable : public IVersioned {
	void FB_CARG dispose();
};
```

Used as base for IStatus

Explicit lifetime control

IRefCounted – OK to pass across thread boundary, can be destroyed by any thread in safe way

```
class IRefCounted : public IVersioned
{
        void addRef();
        int release();
};
```

Base of many interfaces, including IPlugin-Base

IPluginBase

Base interface for all primary plugin interfaces

```
class IPluginBase : public IRefCounted {
    void setOwner(IRefCounted*);
    IRefCounted* getOwner();
};
```

Plays key role when unloading plugin module from process space

IPluginFactory

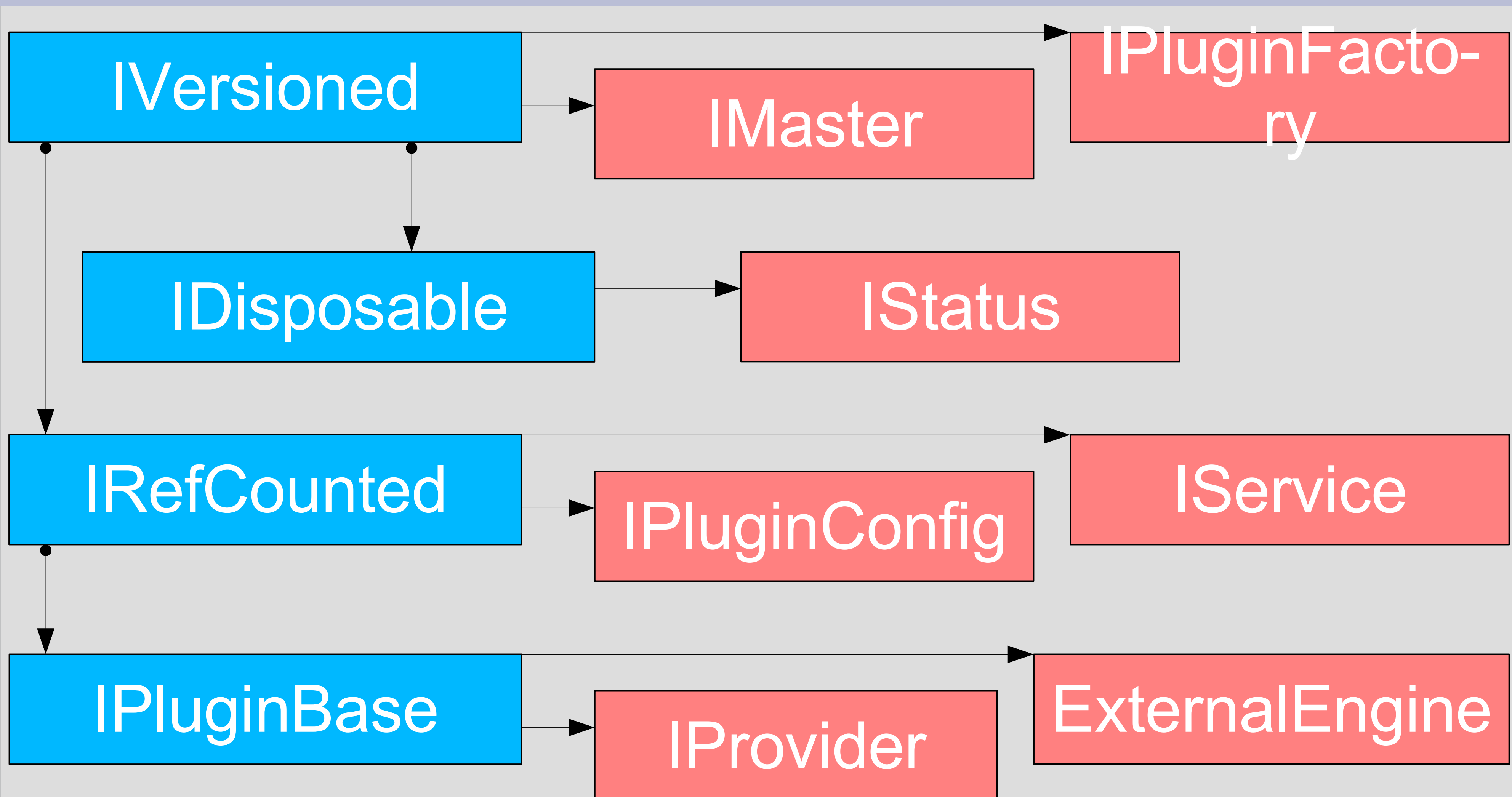
Created by plugin module to register plugin in firebird:

```
class IPluginFactory : public IVersioned {
        IPluginBase* createPlugin
                (IPluginConfig* factoryParameter);
};
```

Registered in plugin manager:

```
IPluginManager::registerPluginFactory
    (int pluginType,
     char* name,
     IPluginFactory* factory)
```

Interfaces hierarchy



C++ wrapper over interfaces

Helps to perform repeating tasks when working with interfaces

```
class Abc: public IVersionedIface { .... }
#define ABC_VERSION (FB_VERSIONED_VERSION + 3)

template <class C, int V> class VersionedIface;

Abc* abcInstance =
    new VersionedIface<Abc, ABC_VERSION>;
```

C++ wrapper over interfaces

Other useful templates:

- template <class C, int V> class AutoIface;
- template <class C, int V> class DisposeIface;
- template <class C, int V> class RefCntIface;
  - missing release() method
- template <class C, int V> class StdPlugin;
- template <class P> SimpleFactory;

Configuring plugins

- List of plugins to be used
  - Set in firebird.conf for each plugin type
  - Default values:

    AuthServer = Srp, Win_Sspi
    AuthClient = Srp, Win_Sspi, Legacy_Auth
    UserManager = Srp
    TracePlugin = fbtrace
    Providers = Remote,Engine12,Loopback

Configuring plugins

- Plugin's specific configuration
  - Depends only upon plugin itself
- Advantages of using standard configuration
  - Saves time/efforts when writing plugin code
  - Plugin's users configure it in familiar manner
- Standard configuration methods
  - File of predefined format [param=value] in predefined place [$(root)/plugins/]
  - Record in common for all plugins file

Configuring plugins

- New configurartion file plugins.conf
  - Has 2 types of records – config and plugin

- Config record – stores plugin-specific data

```
Config = ConfName {
  Param1 = Value1
  Param2 = Value2
}
```

- Plugin record - sets rules of plugin's loading

Configuring plugins

Plugin record format:

```
Plugin = Name {
  Module = /path/to/module
  RegisterName = regName
  Config = ConfName
  ConfigFile = /path/to/file
}
```

Defaults:

```
Plugin = % {
  Module = $(root)/plugins/%
  RegisterName = %
  ConfigFile = $(root)/plugins/%.conf
}
```

Configuring plugins

When we need plugins.conf

- Names conflict in 2 plugins, taken from different places

```
Plugin = Crypt1 {
    Module = $(root)/plugins/Crypt1
    RegisterName = BestCrypt
}
Plugin = Crypt2 {
    #Module = $(root)/plugins/Crypt2
    RegisterName = BestCrypt
}
```

Configuring plugins

When we need plugins.conf

- Use same plugin with different configuration
- Plugin = first {
  - Module = $(root)/plugins/abc
  - RegisterName = abc
  - #ConfigFile = $(root)/plugins/first.conf
- }
- Plugin = second {
  - Module = $(root)/plugins/abc
  - RegisterName = abc
- }

Accessing configuration data from plugin

Configuration is passed to plugin when it is created:

```
IPluginBase* IPluginFactory::createPlugin
    (IPluginConfig* factoryParameter)
```

Typical implementation (in SimpleFactory):

```
IPluginBase* createPlugin(IPluginConfig* fPar)
{
        P* plugin = new P(fPar);
        plugin->addRef();
        return plugin;
}
```

Accessing configuration data from plugin

IPluginConfig layout

```
class IPluginConfig : public IRefCounted {
    const char* getConfigFileName();
    IConfig* getDefaultConfig();
    IFirebirdConf* getFirebirdConf();
};
```

Related method in IPluginManager

```
IConfig* getConfig(const char* filename);
```

Accessing configuration data from plugin

IConfig and IConfigEntry layout

```
class IConfig : public IRefCounted {
    IConfigEntry* find(const char* name);
    IConfigEntry* findValue(char* name, char* val);
    IConfigEntry* findPos(char* name, int pos);
};

class IConfigEntry : public IRefCounted {
    const char* getName();
    const char* getValue();
    IConfig* getSubConfig();
};
```

Accessing configuration data from plugin

Sample

```
IConfig* group(IConfig* iConf, char* entry)
{
    IConfigEntry* ce = findValue(“Group”, entry);
    return ce ? ce->getSubConfig() : NULL;
}

int count(IConfig* iConf, char* param)
{
    int n;
    for (n = 0; iConf->findPos(param, n); ++n);
    return n;
};
```

Accessing firebird.conf from plugin

Accessing global file

```
iPluginMgr->getConfig("$(root)/firebird.conf");
```

Accessing per-database configuration

```
class IFirebirdConf : public IRefCounted {
    int getKey(char* name);
    int asInteger(int key);
    const char* asString(unsigned int key);
};
```

Accessing firebird.conf from plugin

Sample – from secure remote passwords

```
IFirebirdConf* cnf;

SrpManagement(IPluginConfig* par)
    : cnf(par->getFirebirdConf())
{ }

void start(IStatus* status, ILogonInfo* logonInfo)
{
  int dbKey = cnf->getKey("SecurityDatabase");
  char* secDbName = config->asString(dbKey);
```

Using OO provider's API

Main provider's API - IProvider

```
class IProvider : public IPluginBase {
  IAttachment*  attachDatabase (IStatus* status,
     char* fileName, int dpbLength, char* dpb);
  IAttachment* createDatabase(IStatus* status,
     char* fileName, int dpbLength, char* dpb);
  IService* attachServiceManager(IStatus* status,
     char* service, int spbLength, char* spb);
  void shutdown(IStatus* status,
     int timeout, int reason);
};
```

Using OO provider's API

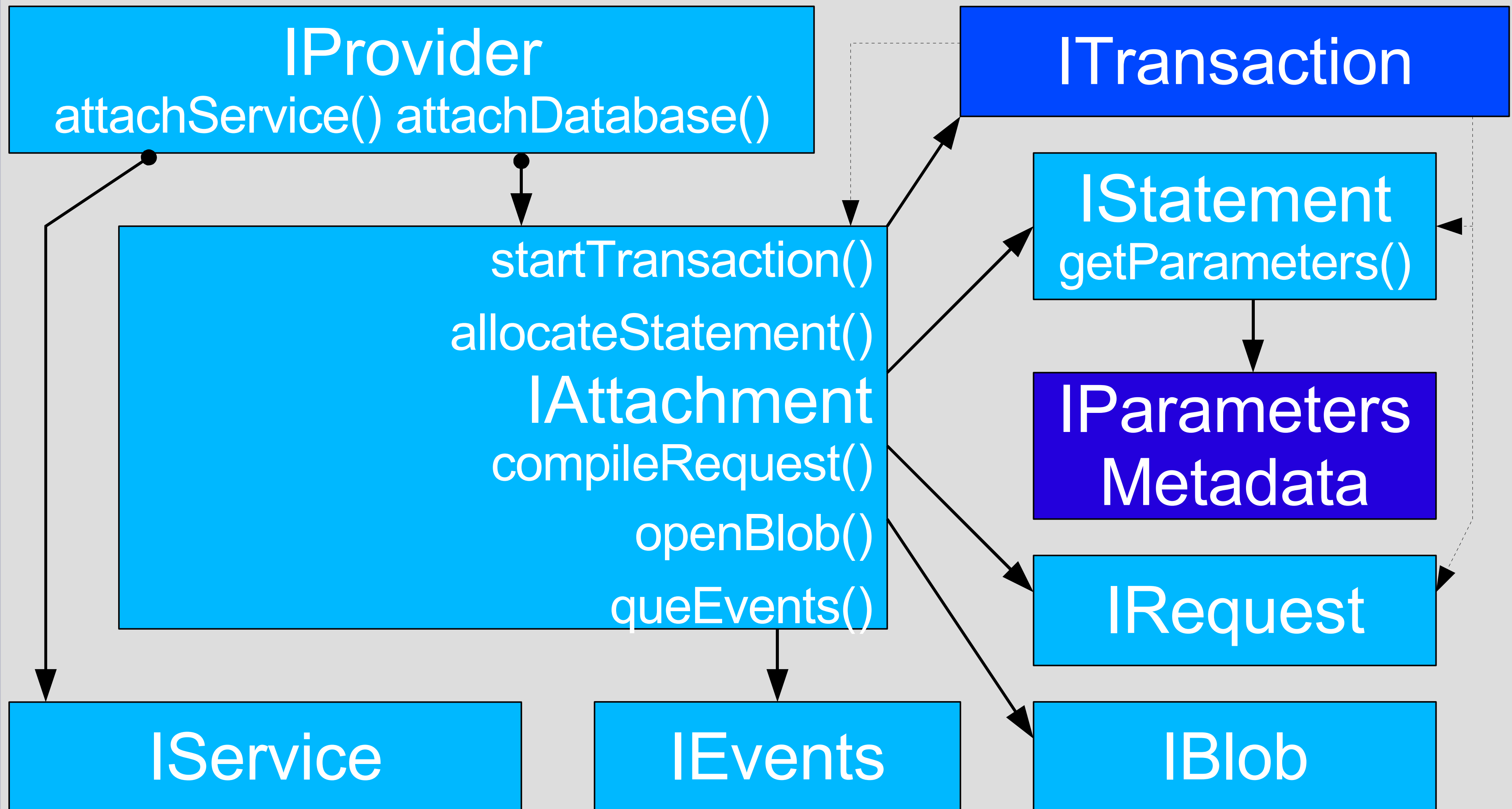
Clients get access to provider's API using IMaster interface:

```
IStatus* status = iMaster->getStatus();
IProvider* dispatch = iMaster->getDispatcher();
IAttachment *att = dispatch->attachDatabase
            (status, “employee”, dpbLength, dpb);
dispatch->release();

// work with attachment ...

att->detach(status);
status->release();
```

Provider's interfaces



Provider's interfaces

IStatement – new message description

class IStatement : public IVersioned {
  int fetch(IStatus* **status, FbMessage* msg);**
  // … other functions
};

isc_dsql_fetch_m(STATUS*, FB_API_HANDLE*,
  USHORT blr_length, SCHAR* **blr, USHORT**
  msg_length, SCHAR* msg);

Provider's interfaces

IEvents – new name, old object

```
class IEventCallback : public Iversioned {
   void callbackFunction(int length, char* events);
};

IEvents* IAttachment::queEvents(IStatus* status,
   IEventCallback* callback, int len, char* events);

class IEvents : public IRefCounted {
   void cancel(IStatus* status);
};
```

Provider's interfaces

```
class IParametersMetadata : public IVersioned {
   int getCount(IStatus* status);
   char* getField(IStatus* status, int index);
   char* getRelation(IStatus* status, int index);
   char* getOwner(IStatus* status, int index);
   char* getAlias(IStatus* status, int index);
   int getType(IStatus* status, int index);
   bool isNullable(IStatus* status, int index);
   int getSubType(IStatus* status, int index);
   int getLength(IStatus* status, int index);
   int getScale(IStatus* status, int index);
};
```

Provider's interfaces

ITransaction – new API functions, related with 2PC (distributed transactions)

```
class ITransaction : public IRefCounted {
   // prepare, commit, rollback, etc...
   ITransaction* join(IStatus* status,
                               ITransaction* tra);
   ITransaction* validate(IStatus* status,
                               IAttachment* attachment);
   ITransaction* enterDtc(IStatus* status);
};
```

Distributed transactions coordinator

Starting transaction in single database

```
IAttachment::startTransaction(IStatus* status,
                              int tpbLength, char* tpb);
```

DTC interface

```
IDtc* iDtc = iMaster->getDtc();
```

Distributed transactions coordinator

DTC interface

```
class IDtc : public Iversioned {
   ITransaction* start(IStatus* status, int cnt,
                       DtcStart* components);
   ITransaction* join(IStatus* status,
             ITransaction* one, ITransaction* two);
};

struct DtcStart {
   IAttachment* attachment;
   char* tpb;
   int tpbLength;
};
```

Distributed transactions coordinator

Sample A

```
DtcStart comp[2] = { {att1, 0, 0}, {att2, 0, 0} };
ITransaction* distr = iDtc->start(status, 2, comp);
```

Sample B

```
ITransaction* t1 =
        att1->startTransaction(status, 0, 0);
ITransaction* t2 =
        att2->startTransaction(status, 0, 0);
ITransaction* distr = t1->join(status, t2);
```

Master interface

```
class IMaster : public IVersioned {
   IStatus* getStatus();
   IProvider* getDispatcher() = 0;
   IPluginManager* getPluginManager();
   int upgradeInterface(IVersioned* toUpgrade,
        int desiredVersion, UpgradeInfo* upInfo);
   ITimerControl* getTimerControl();
   IDtc* getDtc() = 0;
};
```

External Engine

```
class ExternalEngine : public IPluginBase {
   ExternalFunction* makeFunction(Error* error,
               ExternalContext* context,
               IRoutineMetadata* metadata,
               BlrMessage* inBlr, BlrMessage* outBlr);
   ExternalProcedure* makeProcedure(...);
   ExternalTrigger* makeTrigger(Error* error,
               ExternalContext* context,
               IRoutineMetadata* metadata);
};
```

External Engine

Interfaces used

- ExternalContext – attachment of external engine to database.
- IRoutineMetadata – metadata of procedure /trigger /function (name, entry point, etc.)
- Both interfaces passed to plugin from firebird

External Engine

Interfaces exported – procedure, triger and function

```
class ExternalProcedure : public Disposable {
    ExternalResultSet* open(Error* error,
                ExternalContext* context,
                void* inMsg, void* outMsg);
};

class ExternalResultSet : public Disposable {
        virtual bool FB_CALL fetch(Error* error);
};
```

External Engine

Existing plugins:

- UDR – user defined routines on C/C++
  - will be install with firebird server
- JAVA
  - will be isntalled with jaybird

External Engine

JAVA sample – Java class

```
public class FbRegex {
   public static String replace(String regex,
                     String str, String replacement) {
      return str.replaceAll(regex, replacement);
   }
}
```

External Engine

JAVA sample – SQL operator

```
create function regex_replace (
        regex varchar(60), str varchar(60),
        replacement varchar(60)
    )
    returns varchar(60)
   external name
     'org.firebirdsql.example.fbjava.FbRegex.replace (
          java.lang.String, java.lang.String,
          java.lang.String
     )
     return java.lang.String'
   engine java;
```

External Engine

UDR sample – C++ boost-enhanced

```
FB_UDR_BEGIN_PROCEDURE(gen_rows2)
  FB_UDR_EXECUTE_MESSAGE_PROCEDURE
  ( (FB_INTEGER, start)
    (FB_INTEGER, end)
  ,
    (FB_INTEGER, result) )
        { out->result = in->start – 1; }

  FB_UDR_FETCH_PROCEDURE
        { return out->result++ < in->end; }
FB_UDR_END_PROCEDURE
```

External Engine

UDR sample – SQL operator

```
create procedure gen_rows2 (
    start_n integer not null,
    end_n integer not null
) returns (
    n integer not null
)
  external name 'udrcpp_example!gen_rows2'
  engine udr;
```

Trace

```
class TraceFactory : public IPluginBase {
   ntrace_mask_t /*64 bit*/ trace_needs();
   TracePlugin*  trace_create(IStatus* status,
                              TraceInitInfo* init_info);
};

class TraceInitInfo : public IVersioned {
   TraceConnection* getConnection();
   TraceLogWriter* getLogWriter();
   // ..........
};
```

Trace

```
class TraceConnection : public IVersioned {
   int getConnectionID();
   char* getDatabaseName();
   int FB_CARG getProcessID();
// ....
};

class TraceLogWriter : public IRefCounted {
   size_t write(const void* buf, size_t size);
};
```

Also controls audit/trace mode

Trace

```
class TracePlugin : public IRefCounted {
   char* trace_get_error();
   int trace_attach(TraceConnection* connection,
      ntrace_boolean_t create_db,
      ntrace_result_t att_result);
   int trace_transaction_start(
      TraceConnection* connection,
      TraceTransaction* transaction,
      size_t tpb_length, ntrace_byte_t* tpb,
      ntrace_result_t tra_result);
// ....................
};
```

Authentication

3 types of authentication plugins:

- Server – checks whether client correct or not
- Client – prepares data for validation by server
- User manager (not always required)

```
enum AuthResult {AUTH_SUCCESS,
   AUTH_CONTINUE, AUTH_FAILED,
   AUTH_MORE_DATA};
```

Authentication

```
class IAuthServer : public IPluginBase {
	AuthResult  authenticate(IStatus* status,
	    AuthServerBlock* sBlock, IAuthPar* par);
	AuthResult getSessionKey(IStatus* status,
	                  char** key, int* keyLen);
};
class AuthServerBlock : public IVersioned {
	char* getLogin();
	char* getData(int* length);
	void putData(int length, void* data);
};
```

Authentication

```
class IAuthClient : public IPluginBase {
        AuthResult  authenticate(IStatus* status,
             AuthClientBlock* sBlock);
        AuthResult getSessionKey(IStatus* status,
                        char** key, int* keyLen);
};
class AuthClientBlock : public IVersioned {
        char* getLogin();
        char* getPassword();
        char* getData(int* length);
        void putData(int length, void* data);
};
```

Authentication

- 4 plugins in firebird 3
  - Secure remote password protocol
  - Windows - trusted authentication
  - Linux – ssh-like handshake
  - Legacy (DES on client)

Secure remote password

- Up to 20 symbols in password
- Password is never passed over the wire
- Resistant to many attacks, including 'man in the middle'
- Can generate strong crypt keys on both client and server during authentication
- Does not require additional client-server roundtrips due to changes in network protocol

Authentication

```
class IManagement : public IPluginBase {
   void start(IStatus* status, ILogonInfo* lgnInfo);
   int execute(IStatus* status, IUser* user,
                                        IListUsers* callback);
   void commit(IStatus* status);
   void rollback(IStatus* status);
};
```

Authentication

```
class ILogonInfo : public IVersioned {
   char* FB_CARG name();
   char* FB_CARG role();
   int FB_CARG forceAdmin();
   char* FB_CARG networkProtocol();
   char* FB_CARG remoteAddress();
   int FB_CARG authBlock(char** bytes);
};
```

Authentication

```
class IUser : public IVersioned {
   int operation();
   ICharUserField* userName();
   ICharUserField* password();
   ICharUserField* firstName();
   ICharUserField* lastName();
   ICharUserField* middleName();
   ICharUserField* groupName();
   IIntUserField* FB_CARG uid();
   IIntUserField* FB_CARG gid();
   IIntUserField* FB_CARG admin();
   void FB_CARG clear();
};
```

Authentication

```
class IUserField : public IVersioned {
   int entered();
   void setEntered(int newValue);
};

class ICharUserField : public IUserField {
   char* get() = 0;
   void set(char* newValue) = 0;
};
```

Crypt

```
class ICrypt : public IPluginBase {
    void setKey(IStatus* status, int length,
                void* key);
    void transform(IStatus* status, int length,
                void* to, void* from);
};
```

Thanks for your attention!