Creating a P4DTG Plugin:
Integrating Perforce and Defect Trackers (Version 2010.2)

Alan H. Teague, Perforce Software

Overview

The Perforce Defect Tracking Gateway (P4DTG) links job fix information in a Perforce server to defects in a defect tracking system (DTS). P4DTG uses the Perforce jobs system to associate source code changes with defects. When a developer submits a changelist that is associated with the specific job or jobs that it fixes, the fix information is propagated to the associated DTS. When a QA or Technical Support engineer adds a new defect to the DTS, it is propagated to Perforce as a job. Developers can then search for applicable jobs using either the DTS or Perforce Jobs.

The Architecture of P4DTG

P4DTG has two core components: the *configuration tool* and the *replication engine*. Both programs load shared libraries called *plugins* to communicate with Perforce servers and DTS servers. Using the P4DTG Software Developer Kit (SDK), you can create plugins to integrate your defect tracker with Perforce. This paper provides an overview of the process of developing a P4DTG plugin.

The following diagram shows the relationships among the processes and files used by P4DTG.

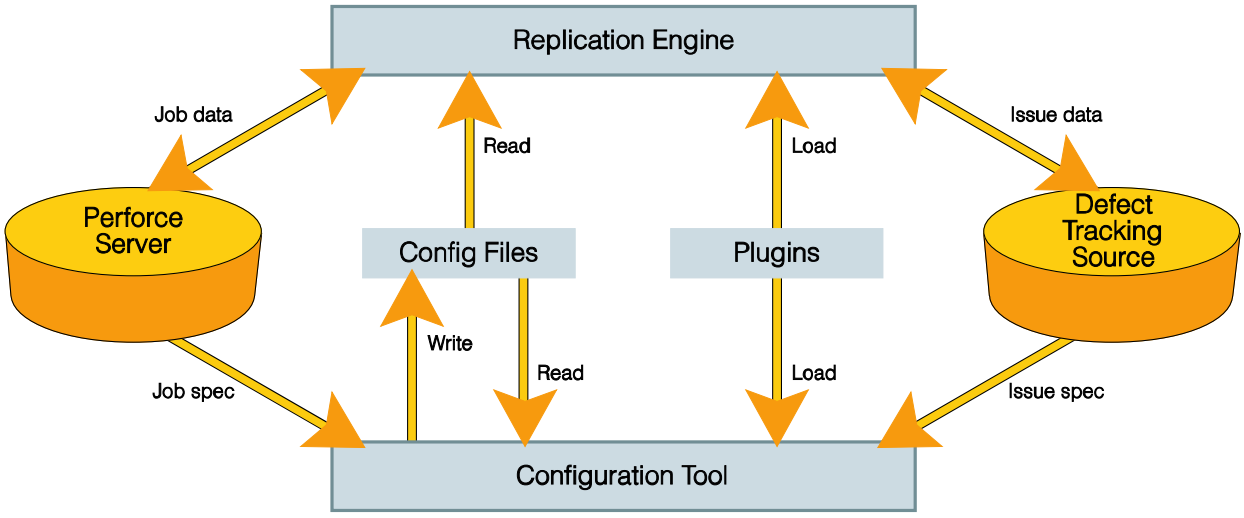

Figure 1: Architecture of the Perforce Defect Tracking Gateway

The Configuration Tool

The configuration tool is a GUI that creates and edits configuration files, which contain the details about how data is replicated. The configuration tool uses plugins to obtain

the metadata from Perforce servers and DTS servers. The metadata specifies the field names and data types for job and defect data.

The Replication Engine

The replication engine uses the plugins to verify the replication mappings and to update jobs and defects during replication. A specific instance of the replication engine links one Perforce server to one DTS server. The DTS server may be another Perforce server running Jobs as a primary defect tracking system.

Configuration Files

The following types of configuration files contain the details for each replication instance:

- **Data source files**: contain the connection details for a specific Perforce server or DTS server, referred to as *data sources* in the configuration tool. A data source can contain one or more *projects*. A project is a set of defects having a common structure and a unique set of defect identifiers. Perforce jobs are treated as a single project.
- **Gateway mapping files**: contain field-to-field mappings that associate job data with defect data for a specified pair of data sources, typically one Perforce server and one DTS server.
- **Status files**: contain the current status and settings for an instance of the replication engine. These settings define the starting time for replication and record when the last replication cycle occurred.

These configurations are stored in XML files on the machine where the configuration tool and replication engine are running. The format of these files is platform-independent.

The Replication Process

When the replication engine is invoked, it performs the following steps:

1. Evaluate every defect in the DTS that was entered or modified between the specified start date and the current system date and time.
2. Make a single pass through newly-updated defects (skipping any updates originated by the dedicated user associated with the engine itself,) checking for updates that occurred while step 1 was being executed.
3. Enter a loop, polling for changes made to defects or jobs by users, replicating the changes as specified by the mappings. This loop repeats until the replication engine is stopped or encounters a fatal error.

Replication Errors

Most errors that the replication engine encounters are not fatal. For non-fatal errors, the job is marked with the error and the replication engine skips that job and its associated defect during subsequent replication cycles. Non-fatal errors are typically one of two

cases: (1) a change made in Perforce violates the workflow in the DTS and the DTS rejects the change; or (2) a **select** field on either side contains old values that are no longer valid and are not mapped. To fix such errors, correct the originating issue or job, then clear the `DTG_ERROR` field in the associated Perforce job. (When correcting jobs, do not connect using the Perforce user that is assigned to the replication engine). The corrected job and issue will be checked and replicated during the next replication cycle.

More serious errors require you to stop the replication engine, correct the errors and then start the engine with the **force** flag set. This flag verifies that all of the jobs and issues have been replicated.

Developing Plugins

For each type of defect tracker, the replication engine requires a plugin. The plugin enables the engine to communicate with defect tracker servers by correctly formatting the required data fields and by defining required methods. (The engine also uses a plugin to communicate with Perforce servers.) Plugins enable P4DTG to be extended to support a variety of defect trackers without requiring changes to the replication engine or the configuration tool.

The following sections tell you how to create plugins.

The Structure of a Plugin

Each plugin for P4DTG must provide a predefined set of functional interfaces, opaque types, and specialized data types. Both the functional interfaces and the specialized data types are C-based to prevent compiler/operating system issues in interacting with a shared library.

The **functional interfaces** are hierarchical, with specific functions that require results from successful calls to outer-level functions. P4DTG requires a plugin to define the following levels of interfaces:

- **Library-level interfaces** exercise the plugin specific functions and do not require a connection to any specific server.
- **Server-level interfaces** are the functions that are available after a base connection to a DTS server has been established.
- **Project-level interfaces** are the functions that are available after a connection to a specific project on a connected server has been made.
- **Defect-level interfaces** are the functions that are available after a specific defect from a specific project on a connected server has been retrieved from the DTS server.

The **opaque types** are essential to the functional interfaces. These types encapsulate the data structure that the plugin uses to identify a connection, a project in a connection, and a specific defect in a project. Both the configuration tool and replication engine use the connection and project objects. Only the replication engine uses the defect objects.

The **specialized data types** contain lists of identifiers (project, job and defect names), dates, field definitions and field name/value pairs. The SDK provides utility functions to assist in the creation, modification and deletion of these types. By using these utility functions, you can avoid the problems caused by mixing calls to the memory allocator functions **malloc** and **new**.

Designing the Plugin

When designing a plugin, consider the following:

- What level of integration is possible?
- What type of workflow is enforced by the DTS? How does that workflow determine which fields must be replicated from Perforce to the DTS? Does your workflow permit the defect tracker to accept defects originating as Perforce jobs?
- How are fields to be mapped? Which fields are to be read-only?
- How must the plugin handle connection authentication?
- Which operating systems and platforms will be supported?
- Can it interoperate with an internationalized Perforce server?
- What must happen when the DTS server goes off-line?
- How will fix details be handled? Does it require a special field for appending?

Levels of Integration

The type of API provided by the DTS determines the level of integration that is possible. A full-function C or C++ TCP-based API provides the best level of integration. If a DTS does not have an external API but the underlying database used by the DTS does, a limited but useful integration is possible. If a DTS uses an interpreted API, a wrapper is required to integrate it into the plugin. One possible approach to creating a wrapper is to provide a TCP-based proxy plug-in that communicates with a secondary service, which is then integrated with the DTS. The “jira” example in the SDK demonstrates this type of wrapper.

Important: If a plugin interacts directly with the database, you are bypassing the business logic contained in the DTS. In this case, you must ensure that the plugin does not update data in ways that violate this business logic. To preserve the workflow, do not allow the plugin to create defects by replicating Perforce jobs or to change the state of defects based on changes to Perforce jobs.

If the defect tracker supports multiple types of databases, you can either encapsulate all of the supported databases using a single plugin (handling the differences internally) or create one plugin for each type of underlying database.

Data Mapping

P4DTG supports the following abstract data types for defect fields:

- `WORD:` single-word text fields

- `LINE:` single-line text fields
- `TEXT:` multiple-line text fields
- `SELECT:` fields that are single selections from a fixed list of values.
- `DATE:` date/time fields
- `FIX:` multiple-line text field (use only as the target for fix details).

The plugin must also provide conversion between `DATE` and the DTS's textual date/time representation.

Authentication

P4DTG supports simple authentication using username and password. More sophisticated authentication can be built into your plugin. Because the replication engine is a background process, the input needed by this type of authentication must be stored outside of P4DTG, for example, in a file in a known location.

Unicode Support

P4DTG supports UTF-8 character strings. All strings passed to and from the replication engine must be in UTF-8 format. Your plug-in can provide an optional interface that controls whether a specific DTS instance can be connected to Unicode-mode Perforce servers. If this interface is not defined, then a warning is generated when connecting to a Unicode-mode Perforce server.

Off-line Behavior

By default, the replication engine waits when it cannot communicate with either the Perforce server or the DTS server. You can control this 'wait-loop' by defining an optional interface that checks whether the server is off-line and, if it is, returns the number of seconds that the replication engine waits before checking again.

The P4DTG Software Developer's Kit (SDK)

The SDK contains a complete C++ framework for building new plugins. This framework is located in the `mydts` directory. You can use it as the basis for plugin development.

The testing tool, `p4dtg-test`, located in the `test` directory, provides a command line interpreter for testing the functional interfaces provided by a plugin. This tool enables you to build and test your plugin incrementally.

Two example plugins are included in the `p4jobs` and `bugs` directories.

- `p4jobs` contains the plugin for Perforce Jobs and shows the use of a DTS's API for building a plugin.
- `bugz` contains the plugin for Bugzilla and demonstrates how a plugin can interact with the database that underlies a DTS.
- `mantis-soap` contains a sample plugin for Mantis and demonstrates how a plugin can interact with a DTS via SOAP web services.
- `jira` contains a wrapper plug-in for implementing a TCP-based XML

request/response system which launches and communicates with a Java-based listener that converts the request into SOAP requests.

The `include` directory contains the header files that define the specific functional signatures for a plugin. The `share` directory contains functions that are useful for building new plugins.

MyDTS Framework Layers

There are three layers in the framework:

- The **exported C functions** for the shared library map the exported interface to the MyDTG class layer. DTG-mod-cpp.cc contains the functions for this mapping. For your plugin, modify the name of the trace file used for protocol-level debugging. Most plugins never need to be debugged at this level.
- The **MyDTG class layer** is composed of three classes: MyDTG, MyDTGProject and MyDTGDefect. Instances of these three classes map to the three opaque types passed back and forth through the shared library interface. These classes provide the base logic for the behavior of these obect types, including the calls to the MyDTS layer and smart caching of intermediate steps.
- The **MyDTS class layer** encapsulates communication with a specific DTS. When developing a plugin, you make most of your modifications to this class.

Error Handling

Plugins communicate errors back to P4DTG using the DTGError structure. This structure contains both the error message and a flag that indicates whether the connection can continue to be used. After a connection is made to a server, P4DTG checks for server warnings. To inform the user of any issues caused by using an older version of the server, plugins can use the `dt_get_server_warnings` interface. This interface is coded in step 3 below.

Coding the Plugin

To code your plugin, perform the following steps.

Step 1: Compile the framework

Compile the MyDTS framework and the test tool. Both of these executables can be compiled without changing any of the code that they contain. After you successfully compile them, run the test tool and issue the LL (Load Library), LT (Library Test), and LF (Library Free) commands. If the module is correctly built, it will successfully load, pass the tests, and be released.

Step 2: Specify plugin name and define date conversion

You must modify the following code:

```
static struct DTGDate *MyDTG::extract_date( const char *date_string );

static char *MyDTG::format_date( struct DTGDate *date );
```

```
static const char *MyDTG::get_name( struct DTGError *error );

static const char *MyDTG::get_module_version( struct DTGError *error );
```

Change the strings that specify the plugin name and version, and code the conversion to and from the `DATE` format used by the DTS. After you make these changes, invoke the test tool and issue the LI, LE, LN, and LV commands (See Appendix A for a listing of the test tool commands and required arguments).

Step 3 (Optional): Specify optional and/or required attributes

If your plug-in has additional attributes that the user can specify when creating a DataSource, modify the following code:

```
static struct DTGAttribute *MyDTG::list_attrs();

static char *MyDTG::validate_attr( const struct DTGField *attr );
```

For `list_attrs()`, create instances of the `DTGAttribute` structure for each attribute for your plug-in. When the user edits an attribute using the configuration tool, the value is checked using a call to `validate_attr(...)`. To indicate success, the validation routine returns `NULL`. If an invalid value is specified, the validation routine returns an error message to be displayed to the user. If an attribute has a default value and the user sets that attribute to that default value, the configuration tool does not save that specific attribute in the configuration file. This approach enables different versions of a plug-in to change default values and apply them to previously-defined data sources.

If your plug-in support Unicode or waiting for offline servers, define attributes that enable the user to configure these features. The MyDTS framework contains the attributes `wait_time` and `unicode`, which provide this configuration.

After you make these changes, invoke the test tool and issue the AL and AV commands.

Step 4: Code connection logic

Mmodify the following code:

```
MyDTG::MyDTG( const char *server, const char *user, const char *pass,
      const struct DTGField *attrs, struct DTGError *error  );

MyDTG::~MyDTG();

struct DTGStrList *MyDTG::list_projects( struct DTGError *error );

char *MyDTG::get_server_warnings( struct DTGError *error );
```

Code the logic required for:

- establishing a connection to a specific DTS
- retrieving a list of projects available for that connection

After you make these changes, invoke the test tool and issue the LC, SF, SL, and SW commands. You can use the AS command to set any supported attributes.

Step 5 (Optional): UTF-8 Support

You must modify the following code:

```
MyDTG::MyDTG( const char *server, const char *user, const char *pass,
      const struct DTGField *attrs, struct DTGError *error  );
int MyDTG::accept_utf8( struct DTGError *error );
```

Code the logic required for:

- saving the value of any Unicode configuration attribute
- returning whether the current connection accepts UTF-8 encoded strings

In the `MyDTG` constructor or in some code called from there, scan for and save any Unicode configuration attributes. In `accept_utf8`, return 0 or 1 depending on whether the connection supports UTF-8 encoded characters. After you make these changes, invoke the test tool and issue the LC, SU, and SW commands. You can use the AS command to set any needed attributes.

Step 6 (Optional): Off-line Server Support

Modify the following code:

```
MyDTG::MyDTG( const char *server, const char *user, const char *pass,
      const struct DTGField *attrs, struct DTGError *error  );
int MyDTG::server_offline( struct DTGError *error );
```

Code the logic required for:

- saving the value of any off-line server configuration attribute
- indicating whether the server is off-line and associated wait-time

In the `MyDTG` constructor or in code called from there, scan for and save any off-line configuration attributes. In `server_offline`, determine if the server is off-line and return the appropriate response. After you make these changes, invoke the test tool and issue the LC, SO, and SW commands. You can use the AS command to set any needed attributes.

Step 7 (Optional): Additional log messages

Modify the following code:

```
int MyDTG::get_message( void *dtID, struct DTGError *error  );
```

Code the logic required for returning any pending messages for inclusion in the replication log. The return value is the log_level for which the message is to be included. A value greater than 3 indicates that no message is to be logged. The message attribute of the error argument is used for the log message.

After you make these changes, invoke the test tool and issue the SM commands. To make testing this interface easier, implement the referenced_fields interface and log a message reporting the fields that are being referenced.

Step 8: Code project logic

Modify the following code:

```
MyDTGProj *MyDTG::get_project( const char *project, struct DTGError *error );
```

```
struct DTGDate *MyDTG::get_server_date( struct DTGError *error );
MyDTGProj::MyDTGProj( MyDTG *dt, const char *project, struct DTGError *error );
MyDTGProj::~MyDTGProj();
```

Code the logic required for:

- connecting to a specific project
- returning server version and date information

After you make these changes, invoke the test tool and run the SP, SD, and PF commands.

Step 9 (Optional): Optimizing defect retrieval

You must modify the following code:

```
void MyDTGProj::referenced_fields( struct DTGStrList *fields );
```

Code the logic required for saving this list of fields for later use in retrieving a defect. These fields are the only fields that are needed for the specific replication and can improve performance of your plug-in by retrieving only essential fields from the DTS.

After you make these changes, invoke the test tool and run the PR command.

You must modify the following code:

```
void MyDTGProj::segment_filters( struct DTGFieldDesc *filter );
```

Code the logic required for saving this definition of the segmentation filter for later use in retrieving a defect. This filter describes the defects that will be accessed during replication and can improve performance of your plug-in by retrieving only essential defects from the DTS. See the 'mydts' and 'p4jobdt' examples for details on implementing this interface.

After you make these changes, invoke the test tool and run the PI command.

Step 10: Define data type mappings

You must modify the following code:

```
struct DTGFieldDesc *MyDTGProj::list_fields( struct DTGError *error );
```

If the underlying API supports queries that return details about the structure of defect data, you can code logic that retrieves that data and maps each retrieved field definition to the appropriate data type and, more importantly, indicates whether the field is writable. One field must contain the date and time that the defect was last changed. Optionally you can designate fields that contain the name of the user who made the last change and the unique identifier for the issue. These fields are designated using special values for the `readonly` element of the `DTGFieldDesc` structure. 2 indicates the modified date field, 3 indicates the modified by field, and 4 indicates the DefectID field.

Your plugin can include special-case logic for specific fields, as required by your workflow.

If the API does not support queries that return details about the structure of defect data, your plugin can either hard-code the mappings or use a local file containing the

mappings.

After you implement this function, invoke the test tool and issue the PS command.

Step 11: Test plugin with configuration tool

After all of the above steps have been completed, test your plugin using the P4DTG configuration tool. Use this tool to verify that you can configure a data store using your plugin.

Step 12: Detect changed issues

Modify the following code:

```
struct DTGStrList *MyDTGProj::list_changed_defects( int max_rows,
      struct DTGDate *since,
      const char *mod_date_field,
      const char *mod_by_field,
      const char *exclude_user,
      struct DTGError *error );
```

Code the logic that retrieves the list of defects that have changed at or after a specified date and time. After coding this function, run the test tool and issue the PL command.

Step 13: Define replication retrieval logic

Modify the following code:

```
MyDTGDefect *MyDTGProj::get_defect( const char *defect,
      struct DTGError *error );
MyDTGDefect *MyDTGProj::new_defect( struct DTGError *error );
MyDTGDefect::MyDTGDefect( MyDTGProj *proj, const char *defect,
      struct DTGError *error );
MyDTGDefect::~MyDTGDefect();
struct DTGField *MyDTGDefect::get_fields( struct DTGError *error );
char *MyDTGDefect::get_field( const char *field, struct DTGError *error );
```

Next, code logic for:

- retrieving specified defects
- creating new defects

You can use the test tool to help you develop this logic incrementally. The PD, PN, DF, DL, and DR commands invoke these interfaces.

For any fields of type FIX, the get_field method must return the appropriate value and set the error message to something like “This field can only be the target of Fix Details”. This logic enables the replication engine to force an exit for a misconfigured mapping.

If the `referenced_fields` interface has been defined, then `get_defect` and `new_defect` can use this information to restrict the data that is retrieved from the DTS.

Step 14: Define replication update logic

Modify the following code:

```
void MyDTGDefect::set_field( const char *field, const char *value,
      struct DTGError *error );
char *MyDTGDefect::save( struct DTGError *error );
```

Code logic that updates data as follows:

- changing specific defect fields
- saving the current defect

Each DTS updates its records by submitting either the entire record or only changed fields. The plugin must encapsulate the behavior that is used by the specific DTS. You can use the test tool to help you implement this logic incrementally. The PW and PS commands invoke these interfaces.

Step 10: Test your plugin!

After you have coded and tested the preceding logic, be sure to test your plugin for memory usage and leaks, using your preferred testing tools.

Conclusion

P4DTG enables you to connect your defect tracker to Perforce. Using the SDK, you can create your own plugins for proprietary defect trackers or those that Perforce does not support directly. When creating your plugin, be sure that its logic and data mappings support your workflow. Use the P4DTG test tool to develop incrementally, testing as you go.

Appendix A: P4DTG Test Tool Commands

To test your plugin, invoke `p4dtg-test`, located in the `test` directory. The following table lists valid commands. To list the commands from the command line, specify the "H" option. To free any allocated objects and exit the program, specify the "Q" option.

Type	Option	Function Tested	Remarks
Attribute	AL	list_attrs()	Returns a list of DTGAttribute
	AV name value	validate_attr(name, value)	Returns error message or NULL
	AS name value		Set an attribute to be passed in on the LC
	AD name		Deletes an attribute
Defect	DF	defect_free(defectID)	Unsets defectID
	DL	defect_get_fields(defectID)	
	DR field	defect_get_field(defectID, field)	
	DS	defect_save(defectID)	
	DW name value	defect_set_field(defectID,name,value)	
Library	LC server user pass	dt_connect(server, user, pass)	Returns a dtID. If no password, specify NULL.
	LE<cr>datestring	extract_date(datestring)	Format of datestring is plug-in specific
	LF	delete DTGModule()	Unloads the library
	LI yyyy mm dd hh mm ss	format_date(date)	
	LL path	new DTGModule(path)	
	LN	dt_get_name()	
	LT	DTGmodule::test()	
	LV	dt_get_module_version()	
Project	PA fix	proj_describe_fix(projID, fix)	Perforce plugin only
	PC defect	proj_list_fixes(projID, defect)	Perforce plugin only
	PD defect	proj_get_defect(projID, defect)	Returns a defectID
	PF	proj_free(projID)	Unsets projID

	PL n d datef userf user	proj_list_changed_defects(projID, ...)	n: max rows to return d: modified since date yyyy/mm/dd/hh/mm/ss datef: ModifiedDate field name userf: ModifiedBy field name user: Exclude user
	PN	proj_new_defect(projID)	Returns a defectID
	PR field1 field2 ...	proj_referenced_fields(projID, ...)	Optional interface
	PI<cr> Field Opt1 Opt2 … .	proj_segment_filters(projID, filters)	Optional interface
	PS	proj_list_fields(projID)	
Server	SD	dt_get_server_date(dtID)	
	SF	dt_free(dtID)	Unsets dtID
	SL	dt_list_projects(dtID)	
	SO	dt_server_offline(dtID)	Optional interface
	SP project	dt_get_project(dtID, project)	Returns a projID
	SU	dt_accept_utf8(dtID)	Optional interface
	SV	dt_get_server_version(dtID)	
	SW	dt_get_server_warnings(dtID)	
	SM	dt_get_message(dtID)	Optional interface