

Document Exchange Protocol 2 (DEP2)

LightComp v.o.s. <info@lightcomp.com>
Petr Pytelka <pytelka@lightcomp.cz>
Karel Žáček <zacek@lightcomp.cz>

Document Exchange Protocol 2 (DEP2)

by , Petr Pytelka, and Karel Žáček

Copyright © 2004-2010 LightComp v.o.s.

Table of Contents

| | |
|---|----|
| 1. Introduction | 1 |
| 2. Changes | 2 |
| 1. Changes 2006-04-25 | 2 |
| 2. Changes 2006-12-18 | 2 |
| 3. Changes 2007-03-28 | 2 |
| 4. Changes 2007-07-26 | 2 |
| 5. Changes 2007-10-02 | 2 |
| 6. Changes 2007-10-09 | 2 |
| 7. Changes 2007-10-18 | 2 |
| 8. Changes 2007-10-22 | 2 |
| 9. Changes 2007-11-07 | 2 |
| 10. Changes 2008-06-24 | 2 |
| 11. Changes 2009-02-17 | 3 |
| 12. Changes 2009-07-14 | 3 |
| 13. Changes 2009-07-21 | 3 |
| 14. Changes 2009-09-14 | 3 |
| 15. Changes 2010-11-04 | 3 |
| 16. Changes 2011-05-09 | 3 |
| 3. Basic Structure | 4 |
| 4. Frame Type 0 | 5 |
| 1. Method Response | 5 |
| 2. Hello | 5 |
| 3. Authorization | 7 |
| 3.1. authorize | 7 |
| 3.2. authorize2 | 7 |
| 4. Connection State | 8 |
| 5. Remote Functions | 8 |
| 6. Objects | 8 |
| 7. Documents | 9 |
| 7.1. Structure Document | 9 |
| 7.2. Push Document | 11 |
| 7.3. Get Document | 11 |
| 7.4. Save Document | 12 |
| 7.5. Remove Document | 14 |
| 8. Files | 14 |
| 8.1. Request File | 14 |
| 8.2. Failed File Request | 15 |
| 8.3. Configuration Request | 15 |
| 8.4. Configuration Request 2 (update configuration) | 15 |
| 8.5. Failed Config Request | 16 |
| 9. Tasks | 16 |
| 9.1. List of Tasks | 16 |
| 9.2. Task Request | 16 |
| 10. Folders | 17 |
| 10.1. Open Folder | 17 |
| 10.2. Close Folder | 17 |
| 10.3. Notifications | 17 |
| 10.4. Rename Folder | 18 |
| 10.5. Add Document to Folder | 18 |
| 10.6. Move Item | 18 |
| 10.7. Remove Folder | 18 |
| 11. Disconnect | 19 |
| 5. Frame Type 1 | 20 |
| 1. File Attributes | 20 |
| 6. Frame Type 2 | 21 |

| | |
|-----------------------------------|----|
| 7. Testing packet | 22 |
| 8. Channel support | 23 |
| 1. Example | 23 |
| 9. Example of communication | 24 |
| 10. Implementation | 25 |

Chapter 1. Introduction

This is specification of Document Exchange Protocol version 2 used in products Document Server, Tahiti and others. Protocol was introduced in year 2004 and underwent several modifications. There is a list of changes in the following chapter. This specification is based on the version from year 2006 (older changes are not tracked in this document).

Original specification was written in Czech Language and was converted to English in 2008. All future versions will be written primarily in English.

Chapter 2. Changes

1. Changes 2006-04-25

- new parameter `int state` in Section 7.3, “Get Document”

2. Changes 2006-12-18

- new method `getTasks`
- new method `requestTask`
- new method `disconnect`
- extension of `errorReceivedDocument`

3. Changes 2007-03-28

- changed return value for `getDocument`

4. Changes 2007-07-26

- new object `Tahiti.System.Message`

5. Changes 2007-10-02

- new API for folders: `openFolder`, `closeFolder`, `setFolder`, `updateItem`

6. Changes 2007-10-09

- new mime-type `Tahiti/Attrs+Bin` for method `pushDocument`
- structure description of frame for mime-type `Tahiti/Attrs+Bin`

7. Changes 2007-10-18

- new methods `removeFolder`, `deleteDocument`, `renameItem`, `moveItem`
- removed method `setFolder`
- changes in folder API: `openFolder`, `closeFolder`, `updateItem`

8. Changes 2007-10-22

- new channel base communication Chapter 8, *Channel support*

9. Changes 2007-11-07

- new method Section 10.5, “Add Document to Folder”
- improved description of Section 10.6, “Move Item”

10. Changes 2008-06-24

- New method Section 8.2, “Failed File Request”

11. Changes 2009-02-17

- New method Section 2, "Hello"

12. Changes 2009-07-14

- New method Section 8.4, "Configuration Request 2 (update configuration)"

13. Changes 2009-07-21

- New method Section 3.2, "authorize2"

14. Changes 2009-09-14

- New method Section 7.4.1.2, "errorStoreDocument"
- SaveDocument structure extension Example 4.2, "SaveDocument structure definition"
- Application specific objects description moved to documentation related to application.

15. Changes 2010-11-04

- New method Section 5, "Remote Functions"

16. Changes 2011-05-09

- Page can be transmitted with attributes. There is new extension FILEATTRIBUTES, Structure Document was extended with optional page attributes.

Chapter 3. Basic Structure

Protocol is based on protocol TCP. It is possible to use raw TCP packets or to use SSL for data encryption. Transferred data are organized in the frames. Frame is base unit for higher protocol layers. Each frame can be used to transfer data file, method call, testing packet etc.

Basic structure of the frame is on following picture:

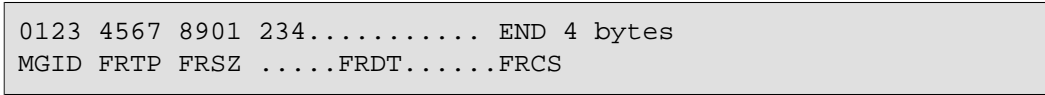


Table 3.1. Structure of the Frame

| Item | Size | Description |
|------|-----------------|--|
| MGID | 4 | Frame Identifier (magic id), expected value 0xE1 0x87 0x05 0xA3 |
| FRTP | 4 | Frame type, number |
| FRSZ | 4 | Size of data in the frame (number of bytes) |
| FRDT | defined in FRSZ | Frame data, size is defined in FRSZ |
| FRCS | 4 | Checksum of the frame, includes MGID,FRTP,FRSZ,FRDT This is not currently used. |

Protocol is initialized in 3 steps:

- Connected - waiting for Hello packet
- Introduced - after hello packet and waiting for authorization
- Authorized

Protocol have to be in Authorized to be able to transfer documents (Section 3, "Authorization"). When protocol is authorized there are no other required steps. However transferred document or file has its own transfer states.

Numeric values (MGID, FRTP, FRSZ, ...) are transferred as LittleEndian (typically used on x86).

Note

Protocol implementation can limit maximal size of the frame (FRSZ). We recommend to limit frame size for security reasons.

Chapter 4. Frame Type 0

Frame type 0 is used for generic remote procedure call - RPC. This method call is based on XML-RPC (s. <http://www.xmlrpc.com/spec>). There is one xml file inside such frame which contains serialized procedure call.

This section contains description of various method which should be implemented. Most of them can be implemented by server and client, some of them only by one side. Method specification is written in pseudo "C" notation.

Every method call contains at least one parameter "ticker" - string. This parameter contains unique method call identification. Such identification can be used to pair together request and response. Every response have to also contain "ticker". Protocol is asynchronous and it is possible to send several independent requests at once.

Example 4.1. Example of method call in XMLRPC

```
<?xml version="1.0"?>
<methodCall>
  <methodName>authorize</methodName>
  <params>
    <param>
      <value><string>ticker</string></value>
    </param>
    <param>
      <value><string>user</string></value>
    </param>
    <param>
      <value><string>password</string></value>
    </param>
  </params>
</methodCall>
```

1. Method Response

Method response is standard response as defined in XML-RPC. Each response has to contain as first parameter "ticker" which has to have same value as "ticker" send by caller. Example of metho response:

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><string>ticker</string></value>
    </param>
    <param>
      <value><i4>130</i4></value>
    </param>
  </params>
</methodResponse>
```

2. Hello

First step is to introduce client to the server. There is method called `hello`.

```

Struct HelloData {
    // Application name
    string systemName;
    // Application version
    string systemVersion,
    // Array of supported options
    array options {
        struct Option {
            // Name of option
            string name;
            // Value of option
            string value;
        };
    };
};

```

```
HelloData hello(string ticker, HelloData data);
```

Table 4.1. hello, parameters

| Parameter | Description |
|-----------|-----------------------------|
| data | structure describing client |

Returns informations about server and its protocol implementation.

Note

For compatibility with older clients server should be able to communicate without `hello` call. Older client will start communication with method `authorize`.

List of options:

| | |
|------------------|--|
| REQUEST_TASKLIST | This is server extension. Server should send it to the client if supports method <code>requestTaskList</code> . Value have to be 1. If server does not support this extension option is not sent or value is set to 0. |
| FOLDERS | This extension requires support on server and client. Both sides have to be notified about folder support. If side does not support this extension option is not sent or value is set to 0. |
| AUTHORIZE2 | This is protocol extension. Server should send it to the client if supports method <code>authorize2</code> . Value has to be 1. If server does not support this extension option is not sent or value is set to 0. Client can use method <code>authorize2</code> only when it is supported by server. |
| STOREDOCUMENT | Store document extension requires support on server and client. Both sides has to be notified about STOREDOCUMENT support. Default value is 0. Value has to be set to 1 when server or client want to use <code>errorStoreDocument</code> notification or send extended error messages in <code>SaveDocument</code> structure. |
| FILEATTRIBUTES | Protocol with this extension can transfer pages with attributes. Both sides have to support this extension to send page with attributes. Detail description is in the chapter Section 7.1, "Structure Document". |

3. Authorization

Next step to establish a connection is to authorized client. This can be done by calling method `authorize` or `authorize2`.

3.1. authorize

Simple authorization by username and plaintext password.

```
int authorize(string ticker, string user, string password);
```

Table 4.2. authorize, parameters

| Parameter | Description |
|-----------|-------------|
| user | user name |
| password | password |

Return value is number. List of error codes is in the following table

Table 4.3. authorize, List of Error Codes

| Error Code | Description |
|------------|--|
| 0 | authorized |
| 1 | wrong name or password |
| 2 | other error in login |
| 3 | rejected by server (user with same user and password is already connected from another computer) |

3.2. authorize2

Extended authorization. Authorize user by username and password encrypted by selected method. (Compatibility notice AUTHORIZE2)

```
AuthorizeResponse authorize2(string ticker, string method, string user, string password);
```

Table 4.4. authorize parameters

| Parameter | Description |
|-----------|---|
| method | authentication method (method of password encryption) |
| user | user name |
| password | password |

Table 4.5. authentication methods

| Method | Description |
|----------------|--|
| plain | plain text password |
| external-token | external token (send token provided by external application as password) |

Return value is structure containing error code and error description. Set `errorCode` to 0 when authorization process was successful.

```
/* AuthorizeResponse description */
struct AuthorizeResponse {
    /* error code 0-OK, >0 - fail*/
```

```

int errorCode;
/* string containing error description message */
string errorDescription;
}

```

4. Connection State

Method `setState` allows to change connection state. Directly after authorization only one way communication is allowed (client to server). Server can send message to the client only when connection is in state '1' (default state after connection is '0').

```
void setState(string ticker, int iNewState);
```

Table 4.6. Set State, possible values

| Value | Description |
|-------|--|
| 0 | default state, method like <code>pushObject</code> and <code>pushDocument</code> are not allowed |
| 1 | allow to receive objects, documents, ... |

5. Remote Functions

It is possible to call remote functions. There is method named `callFunction`.

```
FunctionResponse callFunction(string ticker, string functionName,
string parameter);
```

Table 4.7. call parameters

| Parameter | Description |
|---------------------------|--|
| <code>functionName</code> | name of remote function |
| <code>parameter</code> | Parameter of the function. Format of the parameter depends on the function. JSON serialized object is usually used |

Method `callFunction` return result of the remote function.

```

/* call response structure */
Struct FunctionResponse {
    /* error code 0-OK, >0 - fail*/
    int errorCode;
    /* string with data
    /* if error code if OK, it contains function specific data,
    * usually JSON object
    * if error code >0 then data contains error description
    */
    string data;
}

```

6. Objects

Protocol has API for object transfer. Objects are quite generic mechanism. It is possible to use them for Client to Server and also for Server to Client communication. Each object has a type and set of attributes.

```

/* Object description /
Struct Object {

```

```

/* Object type */
string type;
/* Array of object attributes */
array attributes {
  /* One attribute */
  struct Attribute {
    /* Attribute name */
    string name;
    /* Attribute value */
    string value;
  };
};
};
};

```

Object is send using method `pushObject`.

```
void pushObject(ticker, Object);
```

Method will send object. There is no return value for this method. Value can be returned by another object send in oposite direction.

7. Documents

Document is collection of pages and attributes. Protocol allows to transfer such documents.

7.1. Structure Document

Structure Document is usually used to send/receive document.

```

/* Document description */
Struct Document {
  /* Unique document identifacator */
  string documentId;

  /* Document version. Every document
   * can have more versions. */
  string version;

  /* Document version. This is identification
   * of subversion on server side. Client can
   * safely ignore this value and have to just
   * send it back to server. */
  string serverVersion;

  /* Document type. Document type define which
   * attributes are valid for document, how-to
   * display document and organize in logical
   * structure. Every document have to have
   * valid type. */
  string type;

  /* Document modification.
   * 0 - document is not modified
   * 1 - document attributes are modified */
  int modified;

  /* Document attributes. Array of document
   * attributes. Used attributes depends on document type */

```

```

array attributes {
  /* Structure describes one attribute */
  struct Attribute {
    /* Attribute name */
    string name;
    /* Attribute value */
    string value;
  };
};

/* Array of files, pages. */
array files {
  /* Structure describing one file, page */
  struct File {
    /* Unique file (page) identifiacator */
    string ident;
    /* File mime-type. */
    string mimeType;
    /* Page importance.
     * 0 - page is not important
     * 1 - page is important */
    int important;
    /* Page modification.
     * 0 - page is not modified
     * 1 - page attributes are modified
     * 2 - page data modified
     * 3 - page attributes and data modified */
    int modified;
    /* Page number */
    int pageNumber;
    /* Page notes or comments. */
    string note;
    /* Original filename. */
    string origName;
    /* Keys for page decryption/encryption. */
    string keys;
    /* Page signature. */
    string signature;

    /* Array of page attributes.
     * This array is part of the structure
     * only if protocol extension FILEATTRIBUTES is
     * supported.
     * Attributes are implementation specific and
     * are not defined in this specification.
     */
    array attributes {
      /* Structure describes one attribute */
      struct Attribute {
        /* Attribute name */
        string name;
        /* Attribute value */
        string value;
      };
    };
  };
};
};
};

```

```
};
```

7.2. Push Document

Document can be send from server to client using method `pushDocument`.

```
void pushDocument(string ticker, Document document, int state);
```

Client receiving this method call should display received document. Document can be in read-only or with full access rights.

Table 4.8. pushDocument, parameters

| Parameter | Description |
|-----------------------|--|
| <code>document</code> | Document |
| <code>state</code> | Document state: <ul style="list-style-type: none"> • 0 - document is read-only, no changes can be send back to the server • 1 - document can be modified - read/write access |

Note

Pushed document can have pre-set attribute `modified` and client should properly interpret this attribute.

Note

It is possible to send page without defined mime-type. It is useful when mime-type definition is obtain from the document store together with binary data. Such mime-type is send together with data - look in section Chapter 5, *Frame Type 1*. Instead of real type mime-type `Tahiti/Attrs+Bin` have to be set.

7.3. Get Document

Client can request document using method `getDocument`.

```
GetDocumentResponse getDocument(string ticker, string docId, string version, int state);
```

Table 4.9. getDocument, parameters

| Parameter | Description |
|----------------------|---|
| <code>docId</code> | document identifier |
| <code>version</code> | requested version, empty string means last version is requested |
| <code>state</code> | request mode (0 - read-only, 1 - read/write) |

Method will return structure `GetDocumentResponse`. Requested document will be send using method `pushDocument` (Section 7.2, "Push Document").

```
struct GetDocumentResponse {
    string ticker;
    int retCode;
    string version;
}
```

Item *retCode* is 0 if success. If method failed item *retCode* will be greater then 0.

Item *version* contains returned version (can be used to pair *pushDocument* with this request).

7.4. Save Document

Changed or new document is saved using method *saveDocument*. Save operation is done in three steps:

1. Send new/changed document to the server. Server have to allocate new identifiers and send them to the client.
2. Send binary data files
3. Server have to send confirmation to the client about succesfull save opertaion.

Example 4.2. SaveDocument structure definition

```

/* Structure with newly allocated
 * Identifiers, new version
 */
Struct SaveDocument {
  /* Document state:
   * 0 - ok,
   * 1 - lock,
   * 2 - document not up2date,
   * 3 - error
   * 4 - error with detail, error details are stored
   *     in version and serverVersion fields
   */
  int state;
  /* Document identifier. This is unique
   * document identifier. If new document
   * is saved server has to generate this
   * identifier. When error is returned this field is
   * empty string */
  string documentId;
  /* Identifier of new document version.
   * Each document version has unique combination of
   * docId and version.
   *
   * Error message is stored here when state is 4.
   * This message will be shown in UI to user.
   */
  string version;
  /* Identifier of new server version.
   * Server version is used by optimistic
   * server locks.
   *
   * Detailed error message is stored here when state
   * is 4. This message will be logged on client side
   * and/or shown as detail or error.
   */
  string serverVersion;
  /* List of new Identifiers for files. */
  array<string> NewFileIdents;
};

```

Function for saving new document or new version:

```
SaveDocument saveDocument(string ticker, Document document);
```

Structure Document is parameter of the funcion. All requested items have to be correctly filled in the structure. Document identifier is empty string if document is saved for first time (new document). Server will return new document identifier.

Page identifiers are set only for existing unmodified pages. If page is modified or new identifier is empty string.

Function will return list of new file identifiers and identifier of new document. Server behavioure:

1. If document is requested from the server when newer version is saved last saved version will be returned.
2. If client is trying to save new version of the document in parallel with other client lock error have to be returned. Document is unlock on succesfull save operation, on error or on time-out.

Compatibility notice: error state 4 is valid only when STOREDOCUMENT is set to 1.

Notification about succesfull save operation:

```
void receivedDocument(string ticker, string documentId, string documentVersion);
```

Function has no return value and is used only for client notification.

7.4.1. Error notification

When error occur in the save operation client is notified with method `errorReceivedDocument` or `errorStoreDocument`

7.4.1.1. errorReceivedDocument

```
void errorReceivedDocument(string ticker, string documentId, string documentVersion, int reason);
```

Function has no return value and is used only for client notification. Parametr reason is used to signal error code:

Table 4.10. errorReceivedDocument parameters

| Parameters | Description |
|-----------------|----------------------------|
| documentId | id of document |
| documentVersion | version of document |
| reason | reason of fail, error code |

Table 4.11. reason, list of error codes

| Error Code | Description |
|------------|----------------------|
| 1 | lock |
| 2 | document not up2date |
| 3 | error |

7.4.1.2. errorStoreDocument

```
void errorStoreDocument(string ticker, string documentId, string
version, int reason, string errorMessage, string errorDetail);
```

Function has no return value and is used only for client notification. Compatibility notice STOREDOCUMENT

Table 4.12. errorReceivedDocument parameters

| Parameters | Description |
|-----------------|---|
| documentId | id of document |
| documentVersion | version of document |
| reason | reason of fail, error code |
| errorMessage | error message to be presented in UI. Valid only when reason is 4. |
| errorDetail | detailed error message. Valid only when reason is 4. |

Table 4.13. reason, list of error codes

| Error Code | Description |
|------------|--|
| 1 | lock |
| 2 | document not up2date |
| 3 | error |
| 4 | error with description, parameters errorMessage and errorDetails are used. |

7.5. Remove Document

Existing document can be deleted using method `deleteDocument`. Function has parameter document id as parameter.

```
int deleteDocument(string ticker, string documentId);
```

Function return 0 if success, 1 - document is locked, 2 - no permissions, 3 - other error

8. Files

Methods for file transfer.

8.1. Request File

File from server can be requested using method `requestFile`. When server receive such request it has to send page to the client. File is send in frame type 1 (Chapter 5, *Frame Type 1*).

```
void requestFile(string ticker, string fileId);
```

Table 4.14. pushDocument, parameters

| Parameter | Description |
|---------------|--|
| <i>fileId</i> | File ID (Identifier). This ID have to be unique. |

If file is not available server can send notification `requestFileFailed` (Section 8.2, "Failed File Request").

8.2. Failed File Request

If file request (Section 8.1, "Request File") failed other side should send response `requestFileFailed`. It is up to the caller to correctly display this error to the user.

```
void requestFileFailed(string ticker, string fileId, int errorCode,
string description);
```

Parameter `fileId` is unique file identifier.

Parameter `errorCode` is error code.

Parameter `description` is error description.

8.3. Configuration Request

Configuration file can be requested using method `requestConfig`. Method will send configuration file to the client. File is send as frame, type 2.

```
void requestConfig(string ticker, string fileId);
```

Table 4.15. requestConfig, parameters

| Parameter | Description |
|-----------|--------------------------|
| fileId | configuration identifier |

If file is not available server can send notification `requestFileFailed` (Section 8.5, "Failed Config Request").

8.4. Configuration Request 2 (update configuration)

Configuration file can be updated/requested using method `requestConfig2`.

```
void requestConfig2(string ticker, string fileId, string fileHash,
string hashType)
```

Method will send configuration file if hash of the configuration file on the server is different from the hash of the current configuration file on the client. File is send as frame type 2. Server have to send new configuration file or respond `errorRequestConfig`. Method `errorRequestConfig` is used to signal that current configuration is up-to-date (`errorCode` is set to 0) or to signal wrong request (`errorCode` greater then 0).

Table 4.16. requestConfig2, parameters

| Parameter | Description |
|-----------|--|
| fileId | configuration identifier |
| fileHash | Hash of configuration file stored on client. |
| hashType | Type of function used to compute hash of configuration file. The only possible value is md5 for now. |

Example 4.3. requestConfig2 example

```
requestConfig2("ticker", "config.zip",
"fcd66e666a77a18ca8d08c24f40bc439", "md5");
```

Example requests file `config.zip`.

8.5. Failed Config Request

If configuration file does not exist or some other error occurs, the server should send the following notification about the error.

```
void errorRequestConfig(string ticker, string fileId, int errorCode);
```

Table 4.17. errorRequestConfig, parameters

| Parameter | Description |
|-----------|---|
| fileId | file identifier |
| errorCode | error code <ul style="list-style-type: none"> • 0 - no error, used in requestConfig2 to signal - configuration is up-to-date. • >0 - server specific error |

9. Tasks

Methods for task management.

9.1. List of Tasks

Method `requestTaskList` can be used to request a list of tasks.

```
void requestTaskList(string ticker, array<string> tasks);
```

Parameter `tasks` is a list of currently available tasks on the client.

The server can send a list of tasks to the client using the method `pushTaskList`.

```
void pushTaskList(string ticker, array<string> tasks, int flag);
```

Parameter `tasks` is a list of tasks. Flag values are in the following table:

Table 4.18. flag, List of tasks

| Flag | Description |
|------|---|
| 1 | Full List - this is a full list of all tasks. Client should drop old list and use this one. This is also the response on <code>requestTaskList</code> . |
| 2 | Additional List - tasks in the list should be added to the existing list. |

Total number of displayed tasks on the client can be limited in the server configuration. Tasks which the client sent to the server in the `requestTaskList` and are not part of the response are no longer valid and the client has to delete them from the UI.

9.2. Task Request

A specific task can be requested using the method `requestTask`.

```
void requestTask(string ticker, string taskId);
```

Method `requestTask` from the server. The way of sending a task to the client is up to the server implementation, a common way is to use the method `pushDocument`. The server can notify about an error in a task request using the method `errorRequestTask`.

```
void errorRequestTask(string ticker, string taskId);
```

10. Folders

Documents can be organized in the tree structure. There is API to manipulate with this tree (create, change, remove folders, add items ,..).

Folders are quite similiar to folders on file system. Each folder has name (only root folder is empty string) and contains items. Item is document or another folder.

10.1. Open Folder

Methods allows to open folder. Client will receive for opened folder notifications about changes. Notifications are send until the folder is closed (Section 10.2, "Close Folder").

```
Folder openFolder(string ticker, string folderId);
```

```
struct Folder {
    // Status of folder
    // -3 - no permission to open
    // -2 - folder is locked
    // -1 - folder not exists
    // 0 - ok
    int status;
    // Items in the folder
    array items {
        struct FolderItem {
            // Type
            // 0 - folder, 1 - document
            int type;
            // Item id
            // id of folder or id of document
            string itemId;
            // Item name
            string name;
        };
    };
};
```

10.2. Close Folder

Close folder. Closing folder will also stop sending notifications about folder changes.

```
void closeFolder(string ticker, string folderId);
```

10.3. Notifications

Client (subscriber) will receive notification about folder changes. Following notifications are available:

- new item in the folder
- item was removed from the folder (or item was moved to another folder)
- item was renamed

```
void updateItem(string ticker, string folderId, int oper, FolderItem item);
```

Function has three parameters: path, type of change, item with new data.

Possible types:

0 New item

1 Item was removed (is no longer in the folder)

2 Item was renamed (docId is same with the original)

10.4. Rename Folder

Folder or item can be renamed using method `renameItem`.

```
int renameItem(string ticker, string parentFolderId, string itemId,
string name);
```

`parentFolderId` is id of the folder.

Return values: 0 - ok, 1 - wrong id, 2 - no permissions, 3 - other error

10.5. Add Document to Folder

New document is added to the tree calling `bindDocument`. Document have to be saved before it can be add to the tree.

```
int bindDocument(string ticker, string parentFolderId, string name,
string docId);
```

Return values: 0 - ok, 1 - wrong id, 2 - no permissions, 3 - other error

10.6. Move Item

It is possible to move any tree item to another folder. Method also allows to remove item from the tree.

```
int moveItem(string ticker, string parentFolderId, string itemId,
string newFolderId);
```

`parentFolderId` is id of the original folder

`newFolderId` is id of new folder. If this parameter is empty string item will be removed from the tree.

Return values: 0 - ok, 1 - wrong id, 2 - no permissions, 3 - other error

10.7. Remove Folder

It is possible to delete folder using method `removeFolder`.

```
int removeFolder(string ticker, string folderId);
```

Only empty folder can be removed.

Possible return values:

- 0 - OK
- 1 - wrong folder ID
- 2 - no permission
- 3 - other error.

11. Disconnect

Before disconnect it is possible to notify other side about this event calling method `disconnect`.

```
void disconnect(int iReason, string description);
```

Reason of disconnect is in parameter `iReason`. List of possible reasons is in the table bellow.

Table 4.19. disconnect, parameters

| Constant | Description |
|----------|---|
| 0 | timeout |
| 1 | ukončení aplikace |
| 2 | jiný důvod, popis je uveden v proměnné <code>description</code> |

Chapter 5. Frame Type 1

Type 1 is used to transfer data files. Each transferred file is identified by one identifier (IDDT).

```
FRDT=IDSZ IDDT BIDT
```

Table 5.1. Frame type 1, structure

| Item | Size | Description |
|------|-------------|---------------------|
| IDSZ | 4 | Size of Identifier |
| IDDT | IDSZ | Identifier (string) |
| BIDT | FRSZ-4-IDSZ | Binary data |

It is possible to use special mime-type `Tahiti/Attrs+Bin` for file. Such file contains not only binary data but also XML file with some additional attributes. Embedded XML file contains at least real mime-type of the data file. Structure of such file is bellow:

```
BIDT=XMLSZ XMLDT BDT
```

Table 5.2. Frame, type 1, late attributes

| Item | Size | Description |
|-------|---------------------|----------------------|
| XMLSZ | 4 | Size of the XML file |
| XMLDT | XMLSZ | XML file |
| BDT | FRSZ-4-IDSZ-4-XMLSZ | Binary data |

1. File Attributes

Embedded XML file contains additional attributes for the File.

Example 5.1. Example of File Attributes

```
<?xml version="1.0"?>
<Attributes>
  <Attribute name="Page.mimetype" value="image/tiff"/>
</Attributes>
```

Tag `Attribute` contains name and value of the attribute. Available attributes are in the following table:

Table 5.3. Additional file attributes

| Attribute | Description |
|-------------------|-----------------------|
| Page.mimetype | Mime-type of page |
| Page.origName | Original page name |
| Page.creationtime | Time of page creation |
| Page.importance | Page importance (0 1) |

Chapter 6. Frame Type 2

Type 2 is used to transfer configuration files. Configuration file request is described in section Section 8.3, "Configuration Request". Schema of data blocks with configuration file:.

```
FRDT=IDSZ IDDT BIDT
```

Table 6.1. Description of Frame, type 2

| Item | Size | Description |
|------|-------------|----------------------------------|
| IDSZ | 4 | Size of the identifier |
| IDDT | IDSZ | Configuration File Identifier |
| BIDT | FRSZ-4-IDSZ | Binary data of transferred file. |

Chapter 7. Testing packet

There is testing packet which allows to test connection state. Packet is send by one side and ignored by receiver. Both sides can send this packets.

```
0123 4 bytes MGID
```

MGID is packet Identifier: 0xD0 0x87 0x05 0xA3.

Common practise is to send such packet in regular intervals (tenth of seconds, e.g. 20 sec.) and test if connection is still active.

Chapter 8. Channel support

It is also possible to pack frames into channels. This technique allows to transfer large frames and also manage data transfer. Channels allow to multiplex several frames and transfer them simultaneously. One frame is usually transferred in multiple packets.

```
0123 4567 8901 234...
MGID CHID FRSZ FRDT..
```

Table 8.1. Packet with Channel Support

| Item | Size | Description |
|------|-----------------|---|
| MGID | 4 | Packet identifier (magic id), expected value: 0xF2 0x87 0x05 0xA3 |
| CHID | 4 | Channel ID - same as frame type |
| PKSZ | 4 | Packet size (number of bytes) |
| PKDT | Defined in PKSZ | Packet Data, size is defined in PKSZ |

Transferred data are same as defined for frames. Items MGID, FRTP, FRSC from original frame structure are not transferred. All other items are included.

Note

This feature can be implemented step by step. Easiest implementation is frame type=1.

1. Example

Example of data frame transfer using channels. Frame is packed in two separate packets. Total size of transferred data is 15 bytes..

```
Packet 1:
0xF28705A3 (MGID)
0x00000001 (channel id)
0x0000000A (size of data in the packet)
0x0000000F (size of frame, type 1)
0x12345678 0x1234 (data - 6 bytes)

Paket 2:
0xF28705A3 (MGID)
0x00000001 (channel id)
0x00000009 (size of data in the packet)
0x12345678 0x12345678 0x12 (data - 9 bytes)
```

Chapter 9. Example of communication

There is example of authorization and document request in the following table.

Table 9.1. Example of Authorization and pushDocument

| Direction (C=client, S=server) | Method | Description |
|---|---------------|--|
| C → S | hello | Send hello packet |
| S → C | Ret hello | Response to the hello packet |
| C → S | authorize | Authorization request |
| S → C | Ret authorize | Server response for authorization request |
| C → S | setState | Client change protocol state. |
| S → C | pushDocument | Server sends new document to the client. Clients check its own cache and requests missing pages. |
| C → S | requestFile | Page request |
| S → C | Data, Type=1 | Page data - server response |

Next example shows how-to send changes back to the server.

Table 9.2. Example of Document Save Operation

| Direction (C=client, S=server) | Method | Description |
|---|------------------|--|
| C → S | saveDocument | Save document to the server, server has to create new page ids and send them back. |
| S → C | Ret saveDocument | Server is returning new IDs which will be used to send data. |
| C → S | Data, type=1 | Page data. |
| S → C | receivedDocument | Notification about successfully received document. Notification is sent when whole document is complete. |

Chapter 10. Implementation

This proposal allows quite effective and relatively fast implementation of the protocol. We recommend to use priority queue for the server side implementation. Such queue should allow to send information about documents prior to data files.

There are some aspects of the protocol which can be little bit more complicated to implement:

- Document Save with correct versioning - especially on server-side.
- Received Document - document should be available for users as soon as possible but real data files can be received later.