Internet Group Document Editing Protocol Request for Comments

RFC: 1234

Jason Graalum

jgraalum@pdx.edu

Portland State University

CS 594 – Spring 2018

Internet Group Document Edit(IGDE) Protocol

Status of This Memo

Abstract

The IGDE protocol was developed and implemented over the last 4 weeks as a means for users on the internet to share and edit text documents amongst themselves. Now it supports a building-wide network of servers and clients and is struggling to cope with growth. (I hope and wish!) Over the past 2 days, the average number of users connected to the main IRC network has grown by a factor of 1.001(not really.)

The IGDE protocol is a rich/formatted text-based protocol requiring a simplistic client able to receive real-time updates to the document under consideration as well as a side-band real-time chat.

Table of Contents

1. Introduction

The IDGE (Internet Group Document Editor) protocol has been designed over a number of weeks for use with collaborative rich text-based document editing. This document describes the current IDGE protocol.

The IGDE protocol has been developed on systems using the TCP/IP and UDP/IP network protocols.

IGDE is a distributed text editing system which allows multiple “editors” to concurrently make edits to a rich-text document. The system uses a two-layer client-server topology. A “master” server provides connection, authentication and directory services to “owners” and “editors.” Once an owner has been connected with its editors, the owner becomes the document server for that editing session. Document segments being edited are distributed to the editor clients. If the document owner drops off-line, the editors may continue working on their segments and will delay edit uploads until the owner is back on-line. (Phase II will include an editor taking the roll of document owner to track edits until the owner returns.)

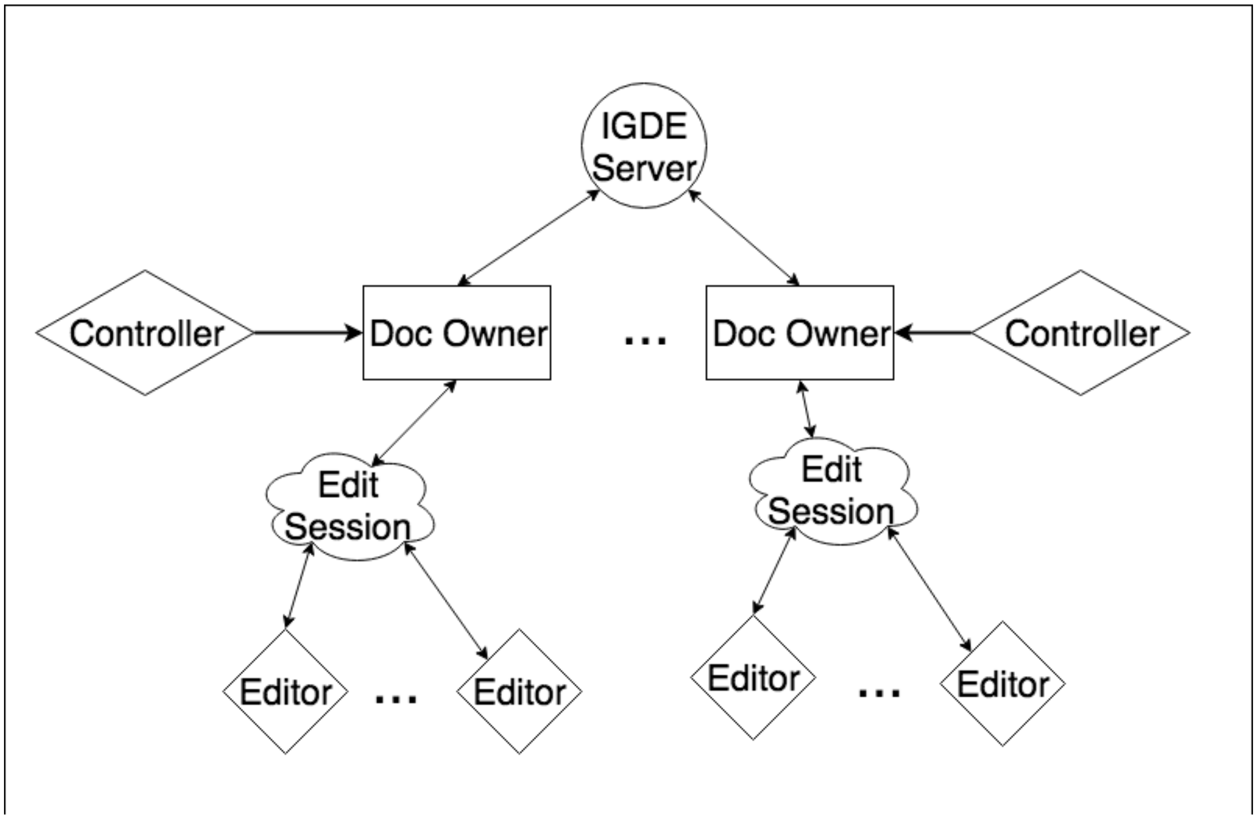

Figure 1. IGDE Organization

1.1. Master Server

The master server provides several main functions. First, it provides authentication and user registration services to track and secure owners and editor accounts. Second, it provides the ability for owners to advertise editing sessions with restrictions on editor access (via authentication). Third, it provides a method for editors to search for owners and owner documents by keeping a list of current edit sessions as well as edit session histories which may be queried by both owners and editors. Finally, the server also gathers statistics on editing sessions and owner/editor activities.

1.2. Owner Client/Server

An owner is both a client and a server. It is a client of the master server where it receives authentication services for itself and its editor clients. It also registers editing sessions with the server in order for editor-client to be able to find the sessions.

When an editing-session is started, it then becomes the server for editor-clients who join the session. As the owner-server, it has two main purposes. First, it controls the document under edit. This includes, the status of all edits (approved, under-review, etc.), current status of all document segments, the history of edits, etc.

The owner-server also tracks the status of all current and past editors. This includes editor identification (via the master server), document permissions (on a segment by segment basis), current cursor position, incoming edits, and past edits. The owner-server also saves the status of all editors when they exit the session. When an editor-client rejoins, the owner-server will reload the editor’s previous state.

1.3. Owner Controller

Each Owner Server has an associated Controller. The controller provides an interactive interface to the Owner Server. The controller also the user to send commands and data to the Owner server such as master server details and edit sessions setup and control. The Controller communicates with the Owner on a unique port different from the Server and Editor connections. The controller port is opened by default when the Owner starts.

1.4. Editor Client

An editor is a client connecting to an editing session via an owner-server with either edit or view-only permissions. Prior to joining an editing session, each editor is authenticated via the master server and is provided owner-server access.

After an editor joins an editing session, it is provided a document segment map by the owner-server. This is similar to a document table of contents. Initially, the document is read-only. The editor must request a segment for edit.

If the editor has previously connected to an editing-session for a specific document, the owner-document will “locate” the editor “cursor” in the previous location.

1.5. Edit Session

An edit-session is initiated by the owner-client after it first connects and authenticates with the master-server. Then the owner-client registers the edit-session with the master-server with a set of attributes regarding the session. These attributes include document title, size and type, allowed editor-clients – either by-name, by-invitation or open-edit.

1.5.1. Edit Session Document

An edit-session document is only fully stored on the owner-client. The master-server has no segments of the document. The editor-clients receive segments of the document when requested(i.e. the editor-client moves the cursor to across a document segment.

A document is divided into segments bases on a maximum segment size(MAX_DOC_SEG_SIZE). An editor-client will have at most two segments depending on where in the document the editor-client's cursor is.

1.5.2. Edit Commands

All edit commands sent by the editor-clients will be based on basic VIM-commands. IGDE will follow the ACNS Bulletin ED-03 "vi Editor "Cheat Sheet" for simplicity. Some examples of editor-client commands that will be implemented (a command prefixed with ':' indicated command mode):

Vi Key	Vi Command	IGDE Code
a	append	a
i	insert before cursor	i
:w	write/commit changes	:w
ESC	exit edit mode	\ESC
h	move left	h
i	move cursor up	i
cw	change current word	cw<text>

Table 1: Edit Commands

Each IGDE code will be either a single 16-bit ASCII character or a combination of up to 8 16-bit ASCII characters. Large command sequences will NOT be implemented in this version.

2. IGDE Specification

2.1. Overview

2.1.1. Component Overview

There are four independent components which make up the IGDE system. There are the master server, the document owner, the controller and the editor. Between these for components, there exists a set of communication channels:

Master	Slave	Purpose
master server	document owner	Register Editing Sessions Advertise to Editors
document owner	controller	Provides an interactive control interface into the document owner
master server	editor	A temporary connection used by the editor to find open edit sessions.
document owner	editor	Provides communication for concurrent editing session between multiple editors

Table 2: IGDE Component Descriptions

2.2. Messaging

2.2.1. Message Overview

Messages are past between the different component via persistent TCP connections. The master-editor connection is open only long enough for the editor to request details on an editing session.

An improvement is proposed to the owner-editor connection which adds a UDP connection for real-time editing commands which are streamed to the owner and back to all other editors. The existing TCP connection sends edit digests which overwrite the UDP editing messages. This provides better interactivity as well as reliable communication.

2.2.2. Message Path Details

There are two types of message paths in the IGDE specification. These are the message loop and receive-reply messaging.

The master server and the document owner use a message loop. This is an “infinite” loop that starts after the process begins. The message loop uses kernel events, which register either socket file descriptors, simple file descriptor and PID signals. In the case of the master server and document owner, each opens its default port and registers the socket as a kernel event. Whenever a signal arrives at this socket, the kernel passes the details to the event handler which signals the loop to continue.

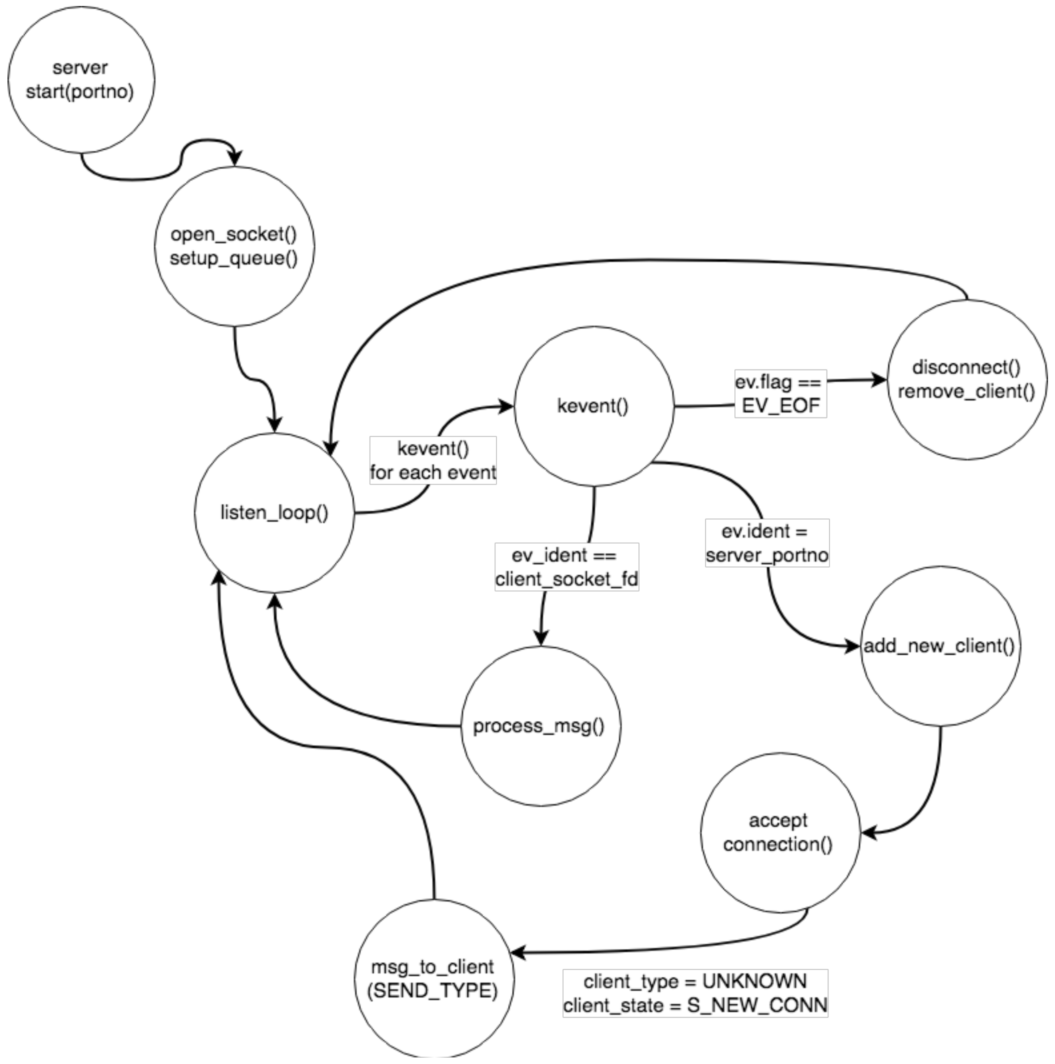
server start(portno)
open_socket() setup_queue()
listen_loop()
kevent() for each event
kevent()
ev.flag == EV_EOF
disconnect() remove_client()
ev_ident == client_socket_fd
ev.ident = server_portno
process_msg()
add_new_client()
accept connection()
msg_to_client (SEND_TYPE)
client_type = UNKNOWN client_state = S_NEW_CONN

Figure 2: Message Loop

The loop takes one of three actions based on the signals included in the event:

Signal	Meaning	
EV_EOF	Source is disconnecting	Remove the source socket from kernel event list.
Identity == master server socket	New client is connecting	Accept connection and add the new socket to kernel event list.
identity == Existing Connection	Current client is sending the server a message	Populate “Message” structure and pass to message processing.
Identity == Controller socket	A controller is trying to connect to the document	If a controller is not already connected, accept and

	owner(This is only for the document owner. The master server does not have an interactive controller.)	register the socket in the kernel event list.

Table 3: Server Signals

Details on the initialization process needed to open the first connections is described below.

2.3. Message Enumeration Details

2.3.1. Server to Owner Signals

```
// Server to Owner Messages
#define SO_BYE                 2001
#define SO_OK                  2002
#define SO_OWNER_TYPE_SET       2003
#define SO_SESSIONS_NOT_OPENED  2004
#define SO_SESSION_CLOSED       2005
#define SO_SESSION_ENDED        2006
#define SO_SESSION_NOT_ENDED    2007
#define SO_SESSION_NOT_REGD     2008
#define SO_SESSION_OPENED       2009
#define SO_SESSION_REGD         2010
```

Figure 3: Server to Owner Signals

2.3.2. Owner to Server Signals

```
// Owner to Server Messages
#define OS_OK                  1000
#define OS_BYE                 1001
#define OS_SET_OWNER_TYPE       1002
#define OS_OPEN_SESSION         1003
#define OS_REG_SESSION          1004
#define OS_END_SESSION          1005
#define OS_CLOSE_SESSION        1006
```

Figure 4: Owner to Server Signals

2.3.3. Owner to Controller Signals

```
// Owner to Controller Messages
#define OC_CLOSE_SESSION       3000
#define OC_DOWNLOAD_CHUNK      3001
#define OC_DOWNLOAD_LAST_CHUNK  3002
#define OC_DOWNLOAD_STARTING    3003
#define OC_EDITOR_LIST         3004
#define OC_ENDING_SESSION      3005
#define OC_SESSION_ENDED       3006
#define OC_SESSION_NOT_ENDED    3007
#define OC_SESSION_NOT_STARTED  3008
#define OC_SESSION_STARTED     3009
#define OC_SRV_CONNECTED       3010
#define OC_SRV_DISCONNECTED     3011
#define OC_SRV_NOT_CONNECTED    3012
#define OC_SRV_NOT_DISCONNECTED 3013
#define OC_UPLOAD_CHUNK_ERR     3014
#define OC_UPLOAD_CHUNK_OK      3015
#define OC_UPLOAD_DONE         3016
#define OC_UPLOAD_FILE         3017
#define OC_UPLOAD_GO           3018
```

Figure 5: Owner to Controller Signals

2.3.4. Controller to Owner Signals

```
// Controller to Owner Messages
#define CO_CMD_FAILED          4001
#define CO_CONNECT_SRV         4002
#define CO_DISCONNECT_SRV      4003
#define CO_DOWNLOAD_CHUNK_ERR  4004
#define CO_DOWNLOAD_CHUNK_OK   4005
#define CO_DOWNLOAD_DONE       4006
#define CO_DOWNLOAD_FILE       4007
#define CO_DOWNLOAD_GO         4008
#define CO_END_SESSION         4009
#define CO_LIST_EDITORS        4010
#define CO_NOT_ACTIVE          4011
#define CO_NOT_CONNECTED       4012
#define CO_OPEN_SESSION        4013
#define CO_SET_CONTROL_TYPE    4014
#define CO_START_SESSION       4015
#define CO_UPLOAD_CHUNK        4016
#define CO_UPLOAD_LAST_CHUNK   4017
#define CO_UPLOAD_STARTING     4018
```

Figure 6: Controller to Owner Signals

2.3.5. Owner to Editor Signals

```
// Owner to Editor Messages
#define OE_OK              5000
#define OE_HELLO           5001
#define OE_BYE             5002
#define OE_INVITE          5003
#define OE_DOC_BLOCK       5004
#define OE_REFRESH         5005
```

Figure 7: Owner to Editor Signals

2.3.6. Editor to Owner Signals

```
// Editor to Owner Messages
#define EO_OK              6001
#define EO_HELLO           6002
#define EO_BYE             6003
#define EO_JOIN            6004
#define EO_CMD             6006
#define EO_RELOAD          6007
```

Figure 8: Editor to Owner Signals

2.3.7. Message Encapsulation

IGDE uses a customer message protocol. Why you might ask! Isn’t this just overkill. Yes. You are right. But, this is a course project, so we don’t all sorts of things just for the fun of learning!

IGDE uses the IGDE_Message class to encapsulate the message details and to provide class methods for packing and unpacking the details into buffers suitable to send to TCP sockets. The IGDE_Message is defined as such:

```
class IGDE_Message
{
private:
    TYPE_T type;
    CMD_T cmd;
    ID_T id;
    TBD_T len;
    char * data;
public:
    int get_msg(int); // Socket FD as input
    int send_msg(int); // Socket FD as input

    int pack(char *);
    int unpack(char *);
};
```

There are some other utility class methods not included here for space considerations. Current, the data types for the type, cmd, id and length message sections are unsigned integer. This results in an IGDE_Message taking 32 bytes in the header plus the data(or payload) size.

type – 4 bytes	cmd – 4 bytes	id – 4 bytes	len – 4 bytes	data - variable

Figure 9: IGDE Message Segments

2.4. Master Server and Document Owner Message Handling

Following initialization, which is described below, the master server and document owers received and replies to all incoming messages from inside there message loops. No out-of-band messaging is allowed as it would interrupt the requirement request-respond sequence which is described below.

2.5. Control and Editor Message Handling

The controller and editor are asynchronous and send messages to the master server and document owners based on user input. (This is following the startup sequence where they make initial connection.)

3. Connection Phases

3.1. Startup

3.1.1. Server

The master server must be the first process to initialize. To initialize, it opens TCP socket, requesters that socket with the kernel event list and then enters its message loop. Prior to entering the loop, the socket must be opened and bound. At this point, the master server is ready to accept connections from both document owners and editors.

3.1.2. Owner

The owner server is similar to the master server with the exception that it is opening a TCP port to listen for Controller connections.

3.2. Connection

When a client initiates a connection to a server(either a controller or editor), it is required to follow the handshake steps in the diagram shown below. This diagram shows the handshake signals and timing for a new owner connecting to the master server. The connection of a new controller to the document owner or a new editor to either the master server or the document owner should follow the same signaling and timing.

3.3. Operation

3.3.1. Server Process

Once the connection is accepted by the server or owner, then all messaging is handled via the message loops. The message loop for the master server processes all incoming message from current clients with the flow shown below. This process shows the document owner states that the server must track to enable correct messaging. This routine is called whenever the server receives a message from an existing document owner.

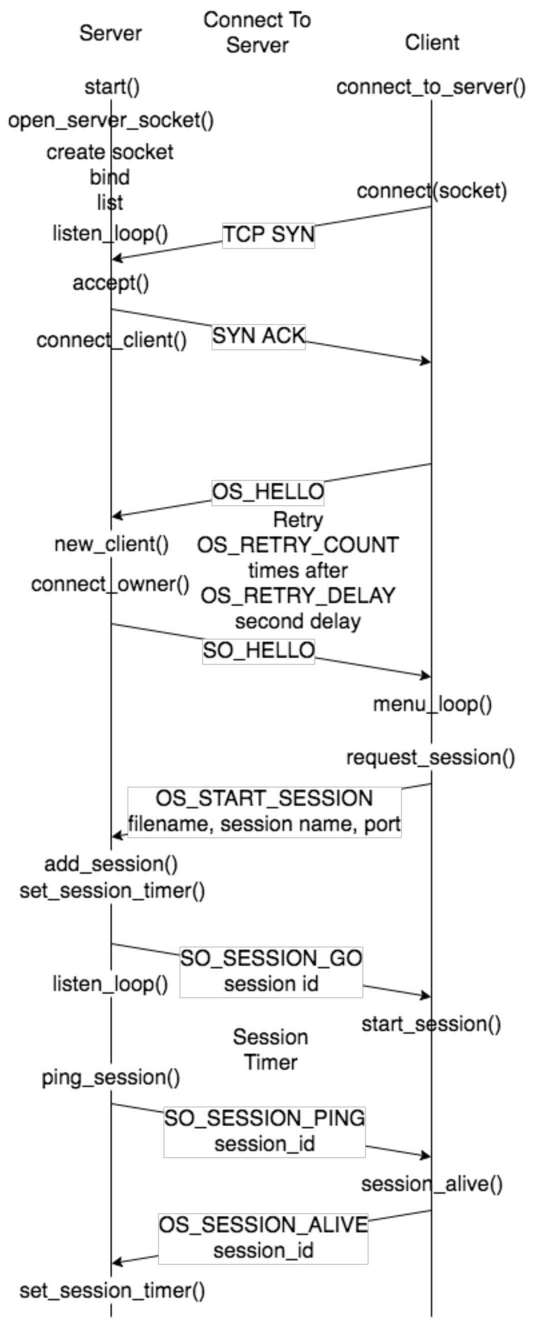

Figure 10: Server Connection Handshake

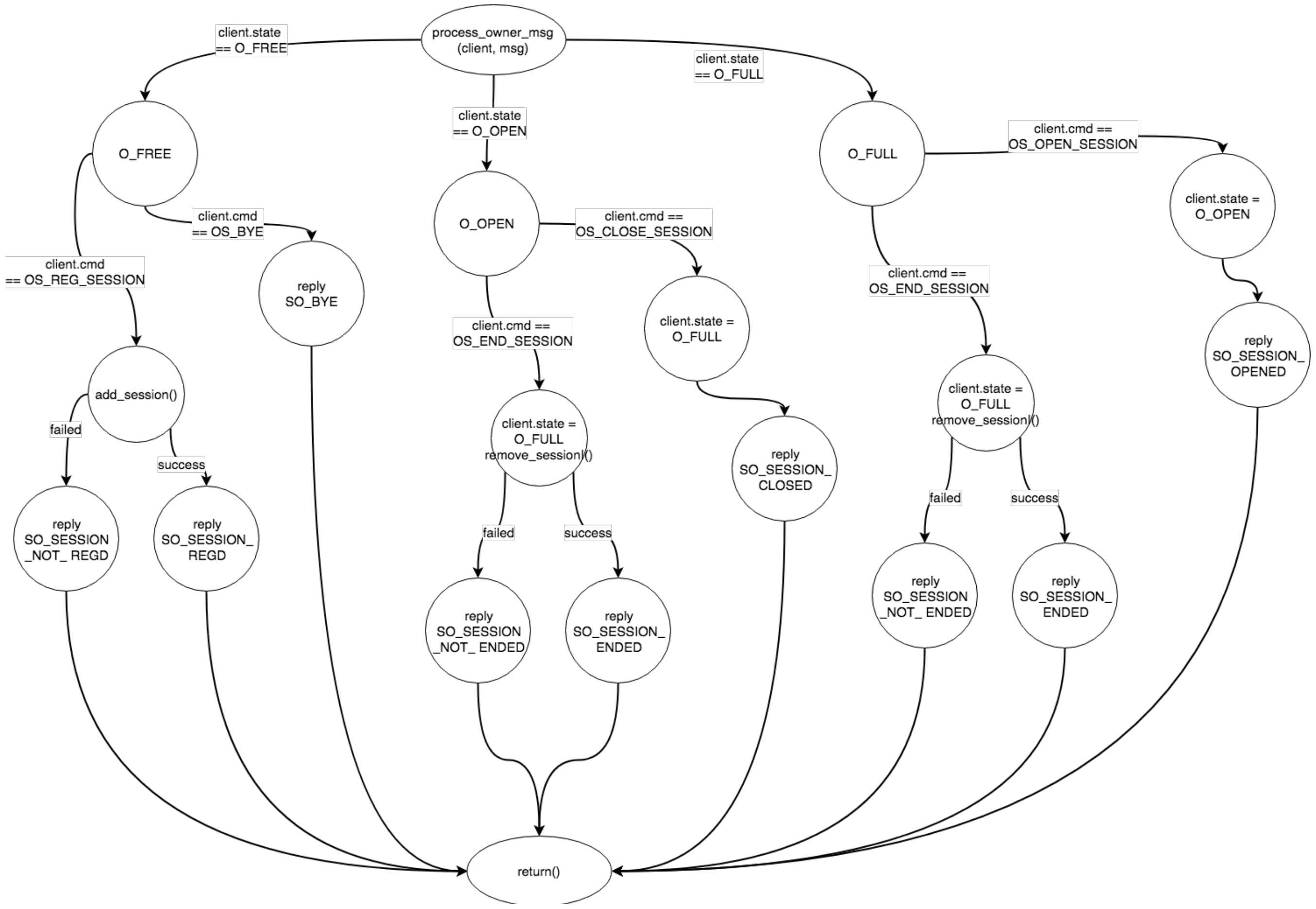

Figure 11: Server Message Processing

3.3.2. Owner Process

The owner process follows a similar process for handling incoming messages. However, it must handles messages from the master server, a controller and incoming editors. All are dealt with out of the document owner message loop – show below.

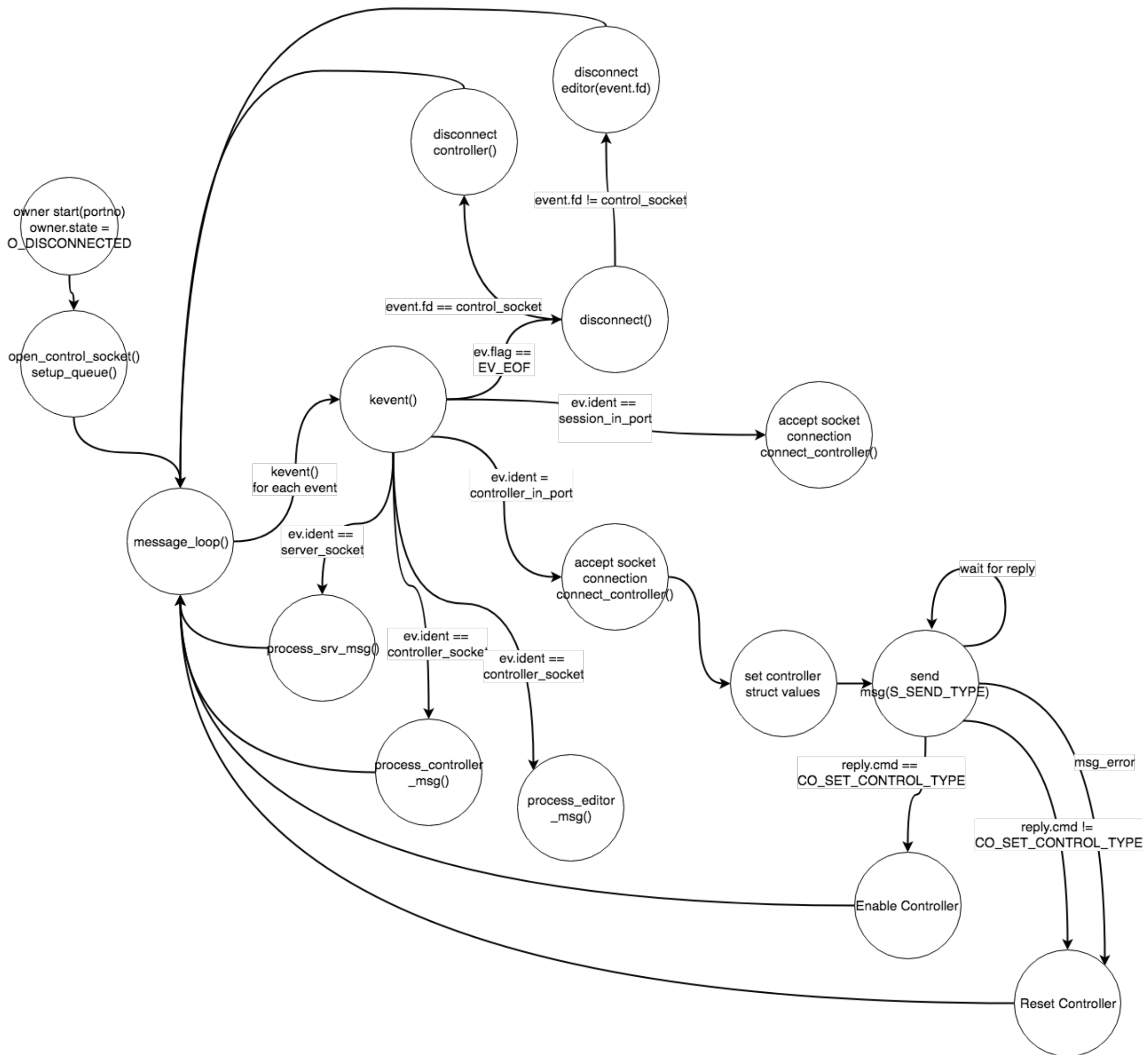

Figure 12: Document Owner Message Processing

3.4. Edit Session Startup

There are a series of communications and handshakes required to start an edit session. These are broken down into four steps:

1. Controller initiates a new session
   a. Provide a name, file and editor port
2. Owner accepts the new session
   a. Allocates temporary Storage
   b. Signals the session to upload the file
3. Session uploads the file chunk-by-chunk
4. Owner verifies the file upload
5. Owner send session registration request to master server

   a. Including details on the session such as filename and maximum number of editor
6. Master Server accepts and registers the new session as “Accepting”
   a. Replies to Owner that the session is Open
7. Owner sets Session state to Accepting

A diagram of the steps is shown below:

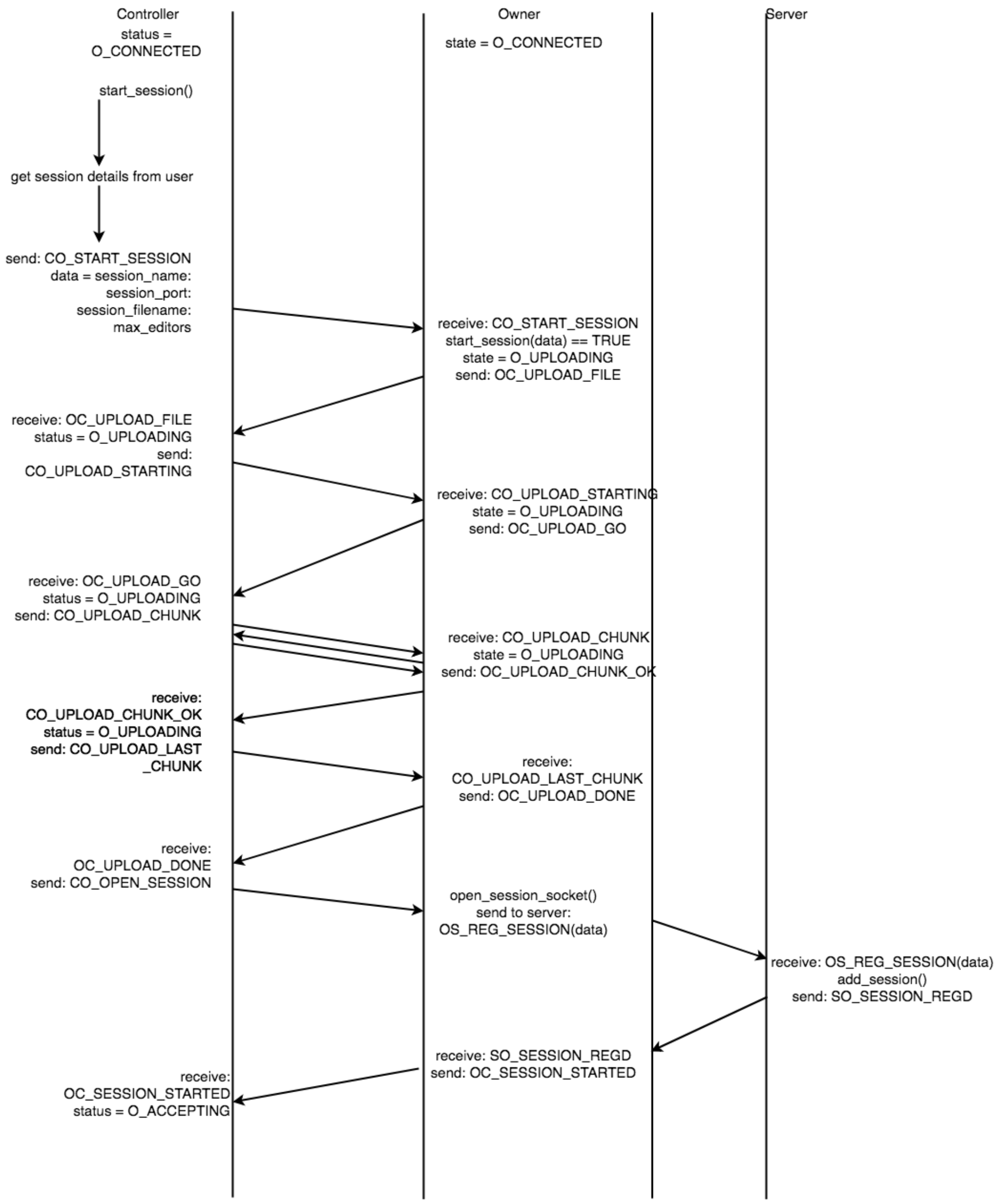
Controller
status = O_CONNECTED
start_session()
get session details from user
send: CO_START_SESSION data = session_name: session_port: session_filename: max_editors
receive: OC_UPLOAD_FILE status = O_UPLOADING send: CO_UPLOAD_STARTING
receive: OC_UPLOAD_GO status = O_UPLOADING send: CO_UPLOAD_CHUNK
receive: CO_UPLOAD_CHUNK_OK status = O_UPLOADING send: CO_UPLOAD_LAST _CHUNK
receive: OC_UPLOAD_DONE send: CO_OPEN_SESSION
receive: OC_SESSION_STARTED status = O_ACCEPTING
Owner
state = O_CONNECTED
receive: CO_START_SESSION start_session(data) == TRUE state = O_UPLOADING send: OC_UPLOAD_FILE
receive: CO_UPLOAD_STARTING state = O_UPLOADING send: OC_UPLOAD_GO
receive: CO_UPLOAD_CHUNK state = O_UPLOADING send: OC_UPLOAD_CHUNK_OK
receive: CO_UPLOAD_LAST_CHUNK send: OC_UPLOAD_DONE
open_session_socket() send to server: OS_REG_SESSION(data)
receive: SO_SESSION_REGD send: OC_SESSION_STARTED
Server
receive: OS_REG_SESSION(data) add_session() send: SO_SESSION_REGD

Figure 13: Session Start Handshake

3.5. Disconnect

Following a complete editing session, the Controller signals to the Owner that the session is ending. At this point, the owner request to send the edited file back to the Controller. A very similar set of setup as the upload are executed, but in reverse with the Owner uploading the file back to the Controller.

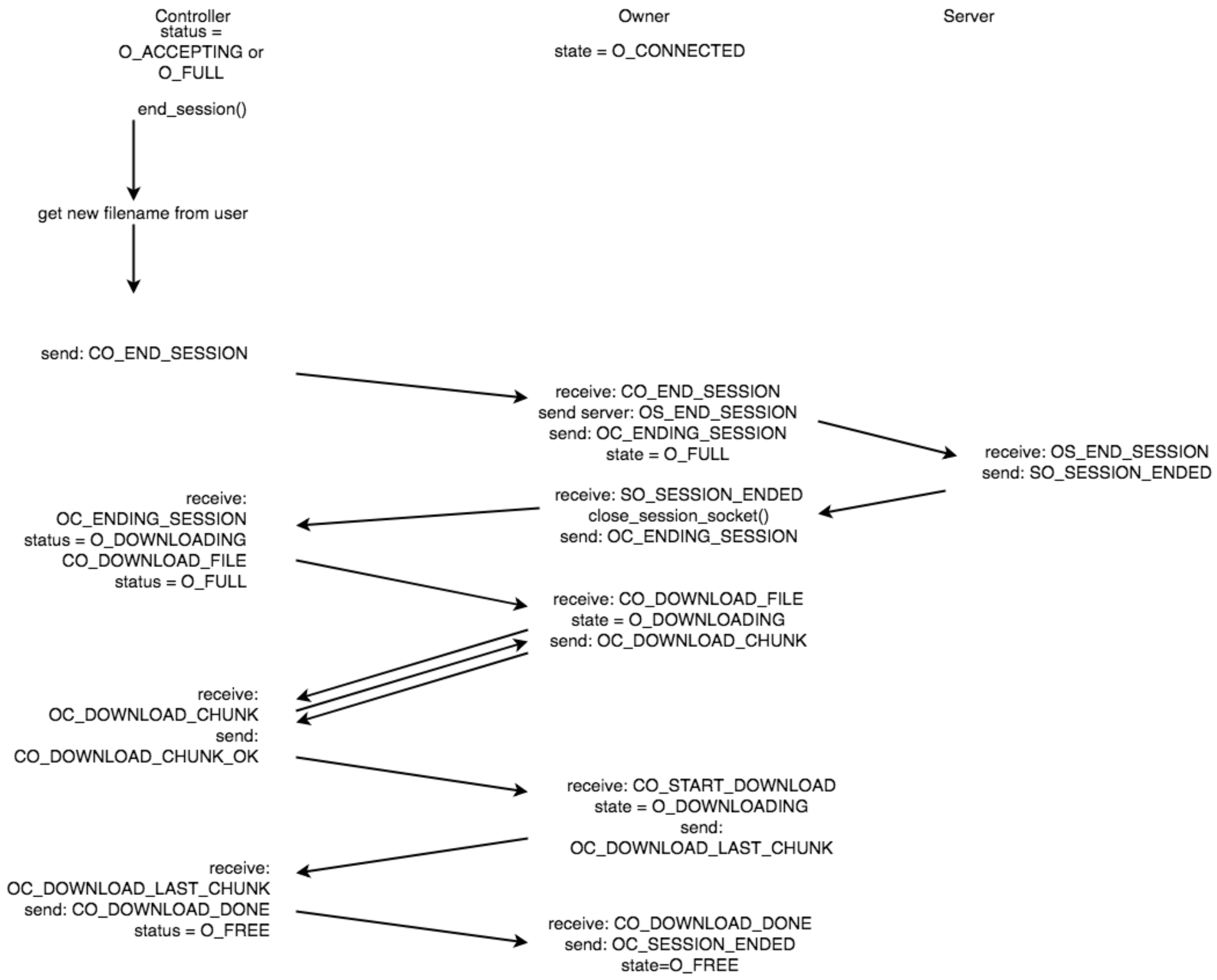

Figure 14: Session End Handshake

4. Current Implementations

The first implementation is a work in progress.

5. Current problems

Beyond count!

6. Current support and availability

No support guaranteed or implied. Use at your own risk!

7. Security Considerations

Too many to list!

8. Author Address

Jason Graalum
5420 NW 146th Ave
Portland, Oregon 97229 U.S.A

Email: jgraalum@pdx.edu