Bump API
Version 1.14
October 11, 2010

1. Introduction

This API lets your program use Bump™ to connect two iPhoneOS applications each running on a separate iPhone or iPod Touch device. After the two devices are connected, a mailbox facility is provided to send chunks of data between the applications. Multiple chunks may be sent, and the connection can persist for some time (e.g. up to 1 hour).

This release (API 1.14) is a production release and supports applications distributed on the App Store.
To upgrade your evaluation key to a production key please visit: http://bu.mp/apideveloper.
The API is distributed as a static library (.a file). It has been tested on iPhone OS 2.2.1 and higher.

-- New in this Version --
Version 1.14 is a bugfix release.
- Fixed a bug where part of the Bump logo was cut off in non US regions.
- Made the main popup text area accommodate up to four lines of text.
- Fixed some French translations.
- Fixed a bump where “Please Bump Again” was not translated.
- Compiled for 4.1 iOS SDK.

Version 1.13 is a bugfix release.
- Fixed a bug where the App could crash after unexpected disconnect.
- Fixed a bug where the close button did not cause the popup to disappear.

--
New in version 1.12 is iPad compatibility and contacts exchange with the main Bump App.
-To support iPad a new config method has been added called `configParentView:`.
You should pass in the view of the view controller where you would like the Bump API popup to be displayed, this way the Bump API popup will follow your view controller’s orientation.
-See section **3.4.1a** of this document for information on sharing contacts with the main Bump App.
-Now supports linking with iOS 4.0

2. What the End-User Sees

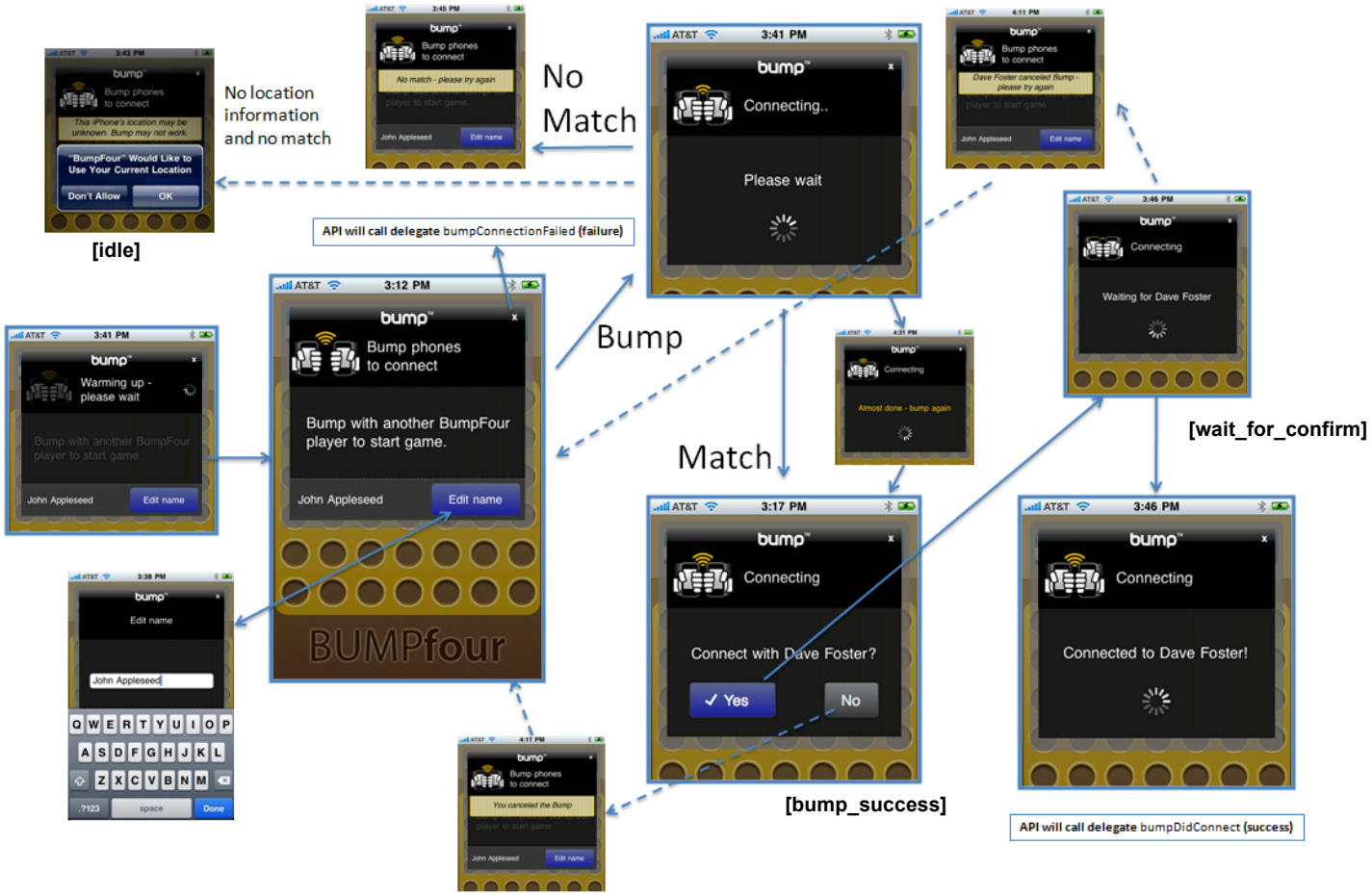
The easiest way to see how the Bump API operates is to get the Adhoc or App Store application BumpFour, which is a two-person version of Connect Four.

In general, the App will have some facility (a button, etc.) to start the Bump connection UI. Bump Technologies will provide different buttons and graphics that can be used (see http://bu.mp/apibumpimgs.html).

To start the Bump API programmatically, a few configuration calls are made, for example to set the API Key (`configAPIKey:`). A `connect*` method is called and a UI (below) pops up to establish the connection. The

`connect*` methods are non-blocking, and a delegate will call back when either a connection has been established or no connection is found.

Here is the user experience flow.



The user experience flow starts with the Bump API popup screen appearing over the current window of the App. The user's name will be pre-populated with their device name, unless the user name has been set with `configUserName:` before `connect` was called. The **[idle]** screen will display the prompt that was set with `configActionMessage:` and then wait for the users to bump phones. If the initial attempt does not result in a successful bump, a number of message banners may be displayed to aid the user. Upon succesful bump, both users will be presented with the **[bump_success]** screen. When a user presses "Yes", they are taken to the **[wait_for_confirm]** screen, and if the other user also confirms, the Bump API popup disappears and control is returned to your Application with a call to `bumpDidConnect`. Control will also be returned to your application at any point during the interaction if the user presses cancel, in this case `bumpConnectFailed:` will be called.

3. API Details

Referencing the header file in Appendix A, the following are details of the required and optional methods and callbacks.

The impatient developer can jump ahead to section 3.3 and 3.4 for more of a how-to style.

3.1 Required & Recommended Methods

3.1.1 Required methods

First:

`-(void) configAPIKey: (NSString *)apiKey;`	Should be called before `connect*`. Pass in the API key that Bump Technologies has provided you.
`-(void) configParentView: (UIView *)parentView;`	optional view where you would like the API popup added. Use this on the iPad so that you can manage the orientation.
`-(void) configActionMessage: (NSString *)actionMessage;`	Used to customize the message (prompting the user to bump) on the idle screen. Highly recommend using this. If not set, the prompt text label will be blank. Example: "Bump with another Texas Hold'em player to start game."

Then:

`-(void) connectToShareThisApp;`	Use the `connectToShareThisApp` method to "bump" your app's iTunes link to the Bump app. The Bump app user will be prompted to download your app from iTunes. This does not use the mailbox facility. Once your App goes live you can update the link to your App on the Bump developer portal.

-or-

-(void) connectToDoContactExchange:(BumpContact*) contact;	Use the connectToDoContactExchange method to "bump" a contact to your App, to another Bump API App implementing connectToDoContactExchange method, or to the main Bump App. Pass in nil if you would like to receive only. More information on this method in the section **sharing contacts**.

-or-

`-(void) connect;`	Use the `connect` method when you will call send:(NSData *)chunk. This will use the mailbox facility for data exchange. You must call `disconnect` so that the Bump server can perform the proper clean up when you're done.
`-(void) send:(NSData *)chunk;`	After a connection is established via the `connect` method, use this to send a chunk of data to the other application. It is improper to use this method if you establish the connection with connectToShareThisApp.

3.1.2 Recommended Methods

`-(void) configHistoryMessage: (NSString *)feed;`	Used to customize the history message displayed on a feed. Example: `@"%1 Started a Texas Hold'em game with %2."` Where %1 is the first players name and %2 is the second.
`-(void) configUserName: (NSString *)name;`	Before a connection is established, use this to set the user name. If called before `connect`, the "Edit name" button will not be shown.
`-(void) disconnect;`	Use this to close the connection to Bump. While this is optional, it is highly recommended when you open the connection with the connect method.

3.1.3 Delegate Methods

`- (void) bumpDidConnect;`	Called when two users have successfully bumped and accepted the interaction. Receiving this means that the system is ready to send and receive data chunks.
`- (void) bumpDidDisconnect: (BumpDisconnectReason)reason;`	Called when either user disconnects. The parameter, `reason`, will contain a value describing the reason for the disconnect i.e. `END_USER_QUIT`, `END_LOST_NET`, `END_OTHER_USER_QUIT` or `END_OTHER_USER_LOST`

`- (void) bumpDataReceived: (NSData *)chunk;`	Called when a chunk of data from the other user has been fetched and is ready to be processed by your application. This is a callback when you use connect and the other device calls send:
`- (void) bumpSendSuccess;`	Called when a chunk of data that you sent via send: successfully makes it to the Bump server for delivery to the other handset. You will receive one callback to this method for each successful send. This doesn't guarantee that the other user has received it.
`- (void) bumpConnectFailed: (BumpConnectFailedReason)reason;`	Called when a `connect*` call results in failure before a connection can be established or before both users have accepted connection. Can result from either user pressing cancel during the interaction or your API key is invalid. The parameter, `reason`, will contain a value describing the reason connect failed – please see Bump.h or Appendix A.
`- (void) bumpContactExchangeSuccess: (BumpContact *)contact;`	Called when a connectToDoContactExchange: call is successful. The contact returned will be filled out with the contact that the other user sent, or will be nil if the other user chose nothing to share.
`- (void) bumpShareAppLinkSent;`	Called when a `connectToShareThisApp:` call is successful in transmitting your apps link to the Bump App. If it fails you'll receive a `bumpConnectFailed:` call instead.

3.2 Optional Methods

`-(NSString *) userName;`	Use this to retrieve the user name after a connection has been established. (Either set by the user on the "setup" screen or previously with the `configUserName:` method.)
`-(NSString *) otherUserName;`	Use this to retrieve the other user name after a connection has been established.

3.3 An Example: How BumpFour uses the API

The BumpFour example has one main file that is important for understanding how the API is used, that is *GameBumpConnector.m*. You'll find *GameBumpConnector.m* in the "Bump API Example" group in the BumpFour Xcode project. The actual implementation of the game logic can be found in the BumpFourGameImplementation group. The curious developer may find BumpFourGameImplementation interesting, but it is not essential to understanding the Bump API interaction.

The underlying game implementation will create a `GameBumpConnector` object, which will in turn create a Bump object and store it in its instance variable, `bumpObject`. When the game is ready to connect to bump it will call `startBump`, which will initiate the bump connection starting the user interaction described in section 2.

GameBumpConnector's `startBump` method:

```
- (void) startBump{
	[self configBump];
	[bumpObject setDelegate:self];
	[bumpObject configHistoryMessage:@"%1 just started a BumpFour match with %2!"];
	[bumpObject configActionMessage:@"Bump with another BumpFour player to start game."];
	[bumpObject connect];
}
```

Once a successful bump has occurred the bump screen will disappear and `GameBumpConnector` will receive a call to its `bumpDidConnect` method, since it is set as `bumpObject`'s delegate. Upon receiving `bumpDidConnect` the `GameBumpConnector` determines the first player, by having both players roll a virtual 1000-sided die and comparing results. The dice rolls are sent back and forth using the `bumpObject`'s `send:(NSData *)chunk` method.

Once a first player has been determined, both clients will send a “startGame” message to the game implementation. The game implementation will begin a loop of sending, waiting for opponent, and receiving moves. When the player, whose turn it is currently, makes a move, `sendGameMove:(int)column` is called on the `GameBumpConnector` object which will package up the move into an NSData object and send it to Bump with `bumpObject`’s `send:(NSData *)chunk` method.

Packaging and sending a move in `GameBumpConnector`’s `sendGameMove:` method

```
- (void) sendGameMove:(int)column{
    if(bumpFourGame.turn == bumpFourGame.localPlayer){ //if it is the local players turn send the move.

        //Create a dictionary describing the move to the other client.
        //We chose to send a dictionary for our communications for this example,
        //But you can use any type of data you like, as long as you convert it to an NSData object.
        NSMutableDictionary *moveDict = [[NSMutableDictionary alloc] initWithCapacity:5];
        [moveDict setObject:[bumpObject userName]  forKey:@"USER_ID"];
        //Tell the other client the action we wish to perform is a "MOVE" so it knows what to do with this data.
        [moveDict setObject:@"MOVE"  forKey:@"GAME_ACTION"];
        //Tell the other client what column we made a move into.
        [moveDict setObject:[NSString stringWithFormat:@"%d", column]  forKey:@"MOVED_COLUMN"];

        //Now we need to package our move dictionary up into an NSData object so we can send it up to Bump.
        //We'll do that with with an NSKeyedArchiver.
        NSData *moveChunk = [NSKeyedArchiver archivedDataWithRootObject:moveDict];
        //[self printDict:moveDict];
        [moveDict release];

        //Calling send will have bump send the data up to the other user's mailbox.
        //The other user will get a bumpDataReceived: callback with an identical NSData* chunk shortly.
        [bumpObject send:moveChunk];
    }
}
```

On the other client an NSData object chunk will be received. In our example we sent a chunk that was a serialized NSDictionary object, so on the receiving end, we’ll unpack the data into another dictionary and act on the result.

Receiving a move in `GameBumpConnector`’s `bumpDataReceived:` method

```
- (void) bumpDataReceived:(NSData *)chunk{
    //The chunk was packaged by the other user using an NSKeyedArchiver, so we unpackage it here with our
NSKeyedUnArchiver
    NSDictionary *responseDictionary = [NSKeyedUnarchiver unarchiveObjectWithData:chunk];
    [self printDict:responseDictionary];

    //responseDictionary no contains an Identical dictionary to the one that the other user sent us
    NSString *userName = [responseDictionary objectForKey:@"USER_ID"];
    NSString *gameAction = [responseDictionary objectForKey:@"GAME_ACTION"];

    NSLog(@"user name and action are %@, %@", userName, gameAction);

    if([gameAction isEqualToString:@"MOVE"]){
        NSString *column = [responseDictionary objectForKey:@"MOVED_COLUMN"];
        NSLog(@"opponent moved to column %@", column);

        [bumpFourGame columnSelected:[column integerValue]];
    }

    .....
}
```

Please refer to the BumpFour example project for a full working example of the Bump API in action.

3.4 Review of How to Use the API

Starting Bump

```
Bump *bumpObject = [[Bump alloc] init];
[bumpObject configAPIKey:@"YOUR_API_KEY"];
[bumpObject setDelegate:self];
[bumpObject configHistoryMessage:@"%1 just started a BumpFour match with %2!"];
[bumpObject configActionMessage:@"Bump another BumpFour app in Bump Mode."];
[bumpObject connect];
```

Implement the following delegate methods to wait for response from `connect:`

```
- (void) bumpDidConnect;
- (void) bumpDidDisconnect:(BumpDisconnectReason)reason;
- (void) bumpConnectFailed:(BumpConnectFailedReason)reason;
```

Once receiving a call to `bumpDidConnect` send data with:

```
[bumpObject send:chunk];
```

Implement the following delegate method to receive data from other client:

```
- (void) bumpDataReceived:(NSData *)chunk;
```

When finished, close the connection with:

```
[bumpObject disconnect];
```

3.4.1a Sharing Contacts With The Bump API

The Bump API v1.1 introduces a new feature in sharing contacts through the Bump API.
The exciting thing about this new feature is that contacts can be shared not only with other copies of your own APP but they can also be sent to and received directly from the main Bump App.

To share a contact you'll first import "`BumpContact.h`" and create and fill out a `BumpContact` object. The next step is to simply call connectToShareBumpObject:(BumpContact*) passing in the contact that you created. If the connection is successful you'll receive a call to your bumpContactExchangeSuccess: delegate method with the contact that the other user sent you (or `nil` if they chose nothing to send).

If the contact that you attempted to send was malformed, you'll receive a `bumpConnectFailed:` call back with the failure reason: `FAIL_BAD_CONTACT` and the API popup will not be shown.

The `BumpContact` object that you fill out and send is a simple object with some properties. All properties in the object are optional but we recommend setting at least one of *first name*, *last name*, or *company name*. Single value properties such as first name, last name, company name, birthday etc, are properties of the object that you can set directly. Properties that support multiple values such as phones and emails are each an `NSArray *` that you should set to be an array of dictionaries filled out in the form specified in BumpContact.h. To send a contact with multiple emails, you'd create an array of these dictionaries and set the `emailAddresses` property of the bump contact to that array. As an example here is the specification for an email record:

From BumpContact.h:

```
/*Email Addresses : An (NSArray *) of (NSDictionary *) objects with key-value pairs:
 {
 BUMP_EMAIL_ADDRESS     ==> (NSString*)addr
 BUMP_FIELD_TYPE  ==> (NSString *)field_type
 }
 */
```

Example of sharing a contact follows…

3.4.1b Sharing Contacts With The Bump API Example Code

The following is an example of sharing a contact with various name fields set, as well as two email addresses. One home address and one work address:

```
//Temp test of send contact
BumpContact *myContact = [[BumpContact alloc] init];
myContact.image = [UIImage imageNamed:@"someimage.png"];
myContact.firstName = @"John";
myContact.middleName = @"T.";
myContact.lastName = @"Appleseed";
myContact.prefix = @"Mr.";
myContact.suffix = @"Esquire";
myContact.companyName = @"Bump";
myContact.department = @"Engineering";
myContact.jobTitle = @"Software Engineer";

NSMutableArray *email_list = [[NSMutableArray alloc] initWithCapacity:3];
NSMutableDictionary* email_addr = [[NSMutableDictionary alloc] initWithCapacity:2];
[email_addr setObject:@"api@bu.mp" forKey:BUMP_EMAIL_ADDRESS];
[email_addr setObject:BUMP_FIELD_TYPE_WORK forKey:BUMP_FIELD_TYPE];
[email_list addObject:email_addr];
[email_addr release];

email_addr = [[NSMutableDictionary alloc] initWithCapacity:2];
[email_addr setObject:@"someone@g.com" forKey:BUMP_EMAIL_ADDRESS];
[email_addr setObject:BUMP_FIELD_TYPE_HOME forKey:BUMP_FIELD_TYPE];
[email_list addObject:email_addr];
[email_addr release];

myContact.emailAddresses = email_list;
[bumpObject connectToDoContactExchange:myContact];
```

4. API Limits

Our API uses long polling while in mailbox mode which means that chunks are delivered to the receiving client as soon as the server has them available. However, this API is currently not suited for real-time games (such as pong) and is better suited towards turn based games and data sharing applications.

In the future there could be limits on data sent or received though the API connection (e.g. 10MB per `connect`) or a time limit for which the API connection can exist (e.g. 1 hour).

Currently there are no limits, but if you plan to exceed 10MB or 1 hour, please let us know.

5. Unsupported Usage

a) Creating and using more than one Bump object in your app has undefined behavior and is not currently supported.

b) While you are connected and using the mailbox facility (via connect, send:), you cannot call connectToShareThisApp (you would need to call disconnect first).

6. Support

For support, please email: api@bumptechnologies.com.

Appendix A. The Bump.h header file. Blue indicates required calls/methods.

```
typedef enum BumpDisconnectReason {
	END_USER_QUIT, //The local user quit cleanly
	END_LOST_NET,  //The connection to the server was lost
	END_OTHER_USER_QUIT, //the remote user quit cleanly
	END_OTHER_USER_LOST  //the connection to the remote user was lost
} BumpDisconnectReason;

typedef enum BumpConnectFailedReason {
	FAIL_NONE,
	FAIL_USER_CANCELED, //The local user canceled before connecting
	FAIL_NETWORK_UNAVAILABLE, //The network was unavailable, and the local user canceled.
	FAIL_INVALID_AUTHORIZATION, //The APIKey was invalid
	FAIL_EXCEEDED_RATE_LIMIT,
	FAIL_EXPIRED_KEY,
	FAIL_BAD_CONTACT //Contact sent via connectToDoContactExchange: was in an unrecognizable format.
} BumpConnectFailedReason;

@protocol BumpDelegate <NSObject>
- (void) bumpDidConnect;
- (void) bumpDidDisconnect:(BumpDisconnectReason)reason;
- (void) bumpConnectFailed:(BumpConnectFailedReason)reason;
- (void) bumpDataReceived:(NSData *)chunk;
- (void) bumpContactExchangeSuccess:(BumpContact *)contact;
@optional
- (void) bumpShareAppLinkSent;
@end

@class BumpHandsView;
@class BumpRoundedRectView;
@interface Bump : NSObject {
	id<BumpDelegate> delegate;
}

@property (nonatomic, assign) id<BumpDelegate> delegate;

// One of the following connect* methods is required
-(void) connect;
-(void) connectToShareThisApp;
-(void) connectToDoContactExchange:(BumpContact *)contact;
-(void) disconnect;

-(void) send:(NSData *)chunk;
-(void) configAPIKey:(NSString *)apiKey;
//optional view where you would like the API popup added.
//Use this on the iPad so that you can manage the orientation.
-(void) configParentView:(UIView *)parentView;
-(void) configActionMessage:(NSString *)actionMessage;
-(void) configHistoryMessage:(NSString *)feed;
-(void) configUserName:(NSString *)name;
-(NSString *) userName;
-(NSString *) otherUserName;
@end
```