Live Update specification

By Nichos

2017.5

Introduction:

Live update includes resource live update and code live update. For now we only consider resource live update. It’s simpler than code live update.

Goal:

I got myself familiar with the live update solutions in Atlantis and Kepler projects. Then I decided to use the solution in Kepler. It’s simpler and enough for our requirements. After some investigation and thinking I found that there are 2 important points that are not totally supported in Kepler’s solution but they are so important that I want to achieve:

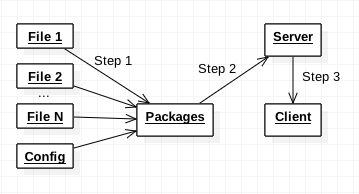
(1) All resource files that are loaded at runtime should be able to do live update. Developers don’t need to plan the live update files in development. After the game is online, we should be able to live update any file we want as long as it’s loaded at runtime. This suggests that to make live update work, we need to make as much resources loaded at runtime as possible.

(2) The solution should be universal. It should support different clients that might use various game engines like cocos2dx and unity, or on various platforms like Android, iOS and Web. So the live update workflow can’t depend on anything special to game engine or platform.

As a result, the server logic is general and doesn’t need to care about what the client is. And all client logics could be almost the same. In implementation perspective, the server code is better to keep as simple as possible to make supporting different clients easier.

Design:

The general live update system is as below:



Step 1: Generate packages

Collect all resource files and some config files and generate one or several packages.

Step 2: Upload packages

Upload packages to server and store them in disk or database.

Step 3: Download packages

Clients download the packages from server.

Step 4: Load packages

Clients load the packages at runtime in some conditions.

Step 1: Generate packages

General design considerations:

(1) The packages should be as less as possible, and be compressed to as smaller size as possible.  
(2) The packages could have Hash or CRC code of the resource files. So clients could verify the packages to ensure the files are not broken or maliciously modified during download.

(3) The packages should include meta information about the resource files. The meta information includes the list of resource files, the hash codes and more possible extensible information in the future.

Implementation:

AssetBundle is the official recommended format for packaging resources in unity. It’s binary and compressed. If in cocos2dx, we can use zip or some other compressed format with high compression rate.

There are 2 steps to generate 2 kinds of asset bundles:

(1) Generate the asset bundles for resource files.

Generally, which asset bundle should one resource file be packaged into is arbitrary. But in practice, we’d better to follow some principles to group the resource files and package them to different asset bundles.

(2) Generate the asset bundle that includes meta information.

When the asset bundles in the first step are generated, it also generates some .manifest files which includes meta information. But the manifest file has more information than we actually need, so we reference the manifest files and generate 2 meta files of our own:

(a) BundleInfo.txt, it includes:

- Resource version number

- List of hash code for every asset bundle of resource files generated in (1)

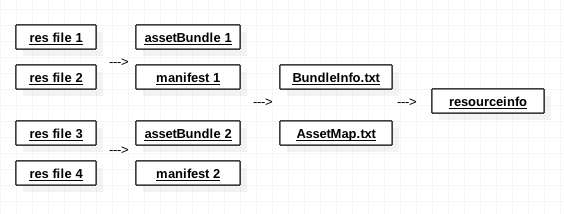
(b) AssetMap.txt, it includes:

- List of mapping a resource name to its asset bundle name

These 2 files are text. So package them to another asset bundle named “resourceinfo” to compress its size.

3 Workflow:

The graph below shows the generation workflow:



In this graph, 3 files will be uploaded to server: assetBundle 1, assetBundle 2, resourceinfo.

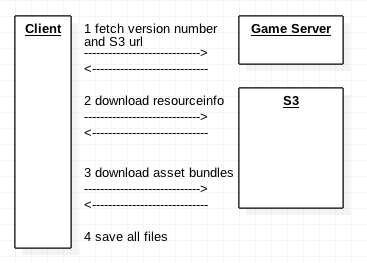
Step 2: Upload packages

In this step, the asset bundles of resource files and meta information are uploaded to server.

A GM tool is needed to conveniently upload the packages. When the packages are uploaded, the server code just simply stores them in disk or database, and records the version number of the resource for client query.

Implementation:  
Server stores the uploaded packages to AWS S3. And it records the version number in Dynamodb.

Step 3: Download packages  
When client starts up, it connects to the game server to download the asset bundles. The graph shows the workflow:



(1) Fetch version number and S3 URL

An example:

(a) The client has the build version 1.1.3. It sends the request with posting its build version 1.1. Only {major.minor} number is needed, the revision number is ignored.

(b) The server checks its database and finds the latest resource version for 1.1 is 1.1.5. Then it responds the number 1.1.5 and S3 URL to the client.

(c) The client receives the resource version 1.1.5 and finds it greater than 1.1.3. So it decides to enter the next step. Otherwise, if the resource version is empty, equal or smaller than the build version, nothing is done and the whole live update workflow ends.

(2) Download resourceinfo

(a) If the client enters this step, it connects to S3 to download resource info file.

(b) After downloading the file, it decompresses it to BuildInfo.txt and AssetMap.txt. In BuildInfo.txt, it lists all asset bundle names of the resource files.

(3) Download asset bundles

Using the list of asset bundles, it downloads all asset bundles of resource files which we actually need.

(4) Save all files

If all the previous steps are successful, it saves all downloaded contents to files for persistence. It includes: BundleInfo.txt, AssetMap.txt, and asset bundles of resource files.

If any step in the process fails, nothing is saved to files. This ensures the download step atomic. Either all files are successfully downloaded and stored, or nothing is changed. Otherwise, if we use part of downloaded asset bundles at runtime, it’s very likely to go wrong.

Step 4: Load packages

When the client loads any resource when running, it doesn’t load the resource file from /Resources folder. Instead, it follows:

(1) Check if there is BundleInfo.txt and if the resource version satisfies the requirements: it has the same {major.minor} number while the revision number is greater than the client build version.

(2) If yes, check if there is the same resource name in AssetMap.txt.

(3) If yes, load it from the downloaded asset bundle. Otherwise, load it from /Resources folder.

Reconsider the goals

Note that how we achieve the goals mentioned in the beginning.

(1) Step 4 ensures any resource file could do live update as long as it’s loaded at runtime. We don’t pre-build asset bundle in development. Instead, whenever the live update is needed when the game is online, package the asset bundles and upload them to server.

(2) We intentionally avoid using the resource cache mechanism in unity. It’s convenient but not available in other game engines like cocos2dx. The whole workflow is universal so it can be applied to any game engine, language and platform.

For the future

There are several improvements which we could do in the future:

(1) BundleInfo.txt includes the hash codes of every asset bundle of resource files. Before trying to download the asset bundles, first check if there is the same asset bundle already downloaded. If yes, compute the hash code and compares it to the hash code listed in BundleInfo.txt. If it’s the same, it means it’s the same file and we don’t need to download it.

This case might occur when we deliver the same files several times during the several live update versions of the same {major.minor}.

(2) In Step 1, generating package is not convenient enough. We could write some scripts to make the process more automated. In Step 2, the packages are also manually uploaded. We could combine these 2 steps to a script for automation.

(3) In Step 2 and 3, an alternative for the server implementation is: the server maintains a file on S3 which lists a table of {major.minor} -> resource version number. In Step 2, every time we upload a package using GM tool, the server simply update the table in the file on S3. Then in Step 3, the client doesn’t need to connect to the game server. It just download the file on S3, parses the content to get the current resource version number.

This would completely free the game server. It needs to serve nothing to client.

(4) After the asset bundles of resource files are downloaded, we could read the content of the file and re-stored it to an uncompressed format, or to a compressed format with lower compression rate but faster loading speed. Currently the asset bundle is lzma compressed format by default. It has high compression rate but is loaded slowly at runtime. So it’s good for network transmission but not good for loading at runtime. This optimization will shorten the loading time.

(5) Delta live update is not supported for now. But I can’t see much sense to do it.

(6) AB test is not supported for now. But based on this system, adding AB test function should be easy.

(7) Any other resource live update solutions?

AB Test

Add on 2017.7.15

Based on the current system, adding AB test is not difficult. The key point is AB test version number adds a suffix character after the normal live update version number:

Normal live update version number format:

1.1.3

AB Test version number format:

1.1.3.a/b/c…

For the 4 steps of normal live update, AB Test has almost the same workflow except these differences:

Step 1:

Generate at least two packages for AB test. You can generate more than two packages if you want to do ABC… test. In each package, use AB test resource version format x.y.z.a as the resource version number in BundleInfo.txt.

Step 2:

When uploading the packages with GM tool, indicate the AB test resource version number to the server. The server stores each AB package in different paths on S3.

Step 3:

On the server side, when the client requests the resource version number, the server logic is:

(1) Check if the AB test is started. If no, continue the normal live update logic. If yes, continue as below.

(2) Decide which a/b version should be responded to the client. This logic is not just simply randomly roll a/b version when a client requests, but has a lot of potential uses. And we can conveniently adjust the logic anytime since it’s on server side. For example, we want to test a/b resources but resource A has very aggressive data tweaks so it’s risky to push it to a lot of users. Then the server could only select a small group of users such as 10% to use resource A and push resource B to other users.

Another point is, the server should store a table mapping client device id to a/b resource version number. To ensure the analytics data we collect accurate, we should make a particular device id bind to either A or B in a test round. But the client might delete the app and reinstall it and this operation shouldn’t affect the A or B group the user is already in. So storing this mapping information in client is not reliable and it should be stored on server side. In a test round, every time a client requests AB version number, the server first checks if the client device id is already in the mapping table. If yes, return queried the AB version number. Otherwise, allocate a new AB version number. If AB test round is restarted, the mapping table is cleared.

(3) Respond the AB version number to the client.

On the client side, the difference is:

(1) When downloading all the files from S3, the URL is concatenated with AB version number other than normal live update version number.

(2) The client should support both AB test and normal live update. Consider the case:

(a) The client downloads normal live update resource of version 1.1.3 first and uses it for some time.

(b) Some time later, it downloads AB test resource of version 1.1.4.a. Now it uses AB test resource and ignores 1.1.3 resource.

(c) Some time later, the client get the response that AB test 1.1.4.a is ended. So it reverts back to 1.1.3 resource and continues using it.

The client should handle this case as we expect. So we store both the AB version number and normal live update version number separately and don’t overwrite one to another. As a result, when AB test ends, the normal live update version can still be read and used.

Step 4:

It’s obvious that AB test has higher priority than normal live update. So the resource version number should be AB test version number if it exists. Otherwise, it should be normal live update version number. Currently, we don’t support use normal live update resource and AB test resource simultaneously. Only one version number is valid and one version of resources is used at one time. This is the simplest implementation but is not convenient in some cases. For example, based on the current implementation, if we upload an important live update fix now and then do AB test later, we need to include the fix package both in AB test packages. But if we support live update and AB test simultaneously, we wouldn’t need to care about the previous live update fix in the following AB test.

Machine Assets

Add on 2017.7.16

For SLOTS games, to make the size of the app bundle small, it’s typical not to package a lot of machine resources in the bundle but place them on server. When the user starts to play this machine in game, the client starts downloading the resources of this machine from server. So machine assets download is a variation of live update and it has some differences as below:

Step 1:

As the basic implementation of this feature, we don’t need to generate BundleInfo.txt, AssetMap.txt and resourceinfo. The only thing is to generate one Asset Bundle for each machine. But there is a disadvantage of stripping off the meta data files: we don’t have any version control mechanism on the machine assets. As a result, once the user downloads the assets for one machine, there would be no chance to re-download the assets in the future, unless the user deletes the app and reinstall it. So if we have improved some machine assets or fixed some visual defects and re-upload them to the server, we can’t push them to users who have already downloaded the old assets. From this point of view, the version control logic as in live update and AB test is still needed and we should add it in the future.

Step 2:

Since there is no meta information to upload to the server, we don’t need GM tool to upload the assets. We only need to manually upload the assets to S3, and the path of each machine asset follows some rules that the client also follows. Each machine asset has different downloading URL for debug and release version, so it could be boring to manually upload them every time the machine assets change. We’ll discuss how to automate the process with some scripts later.

Step 3:

Since there is no version control logic for now, the downloading process is simplified. The client just constructs the machine assets URL according to machine names and downloads them. The host name of URL is configured in Excel file, which can be normally live updated.

Step 4:

Since there is no version control logic for now, the loading process is also simplified:

(1) When clicking the machine button, check if the resource of this machine should be downloaded from server. If no, just load the resources from /Resources folder. Otherwise, continue as below.

(2) Check if the machine asset is downloaded. If yes, load the downloaded resources and enter the machine. Otherwise, start downloading it from S3.

(3) When the downloading succeeds, enter the machine automatically.

As we discussed before, some version control mechanism is still necessary in the future.

Automation

Add on 2017.7.16

For all normal live update, AB test, and machine assets, step 1 and 2 need a lot of boring manual work and waste a lot of time. It should be automated as much as possible. The automation includes:

(1) Editor code to build Asset Bundles.

All configurations can be configured in editor for convenience.

(2) Scripts to upload the Asset Bundles to server.

(a)For normal live update and AB test, some version information is needed, so GM tool is needed and we still manually upload the resources. Keeping some manual work is also good for security.

(b) For machine assets, no meta information is needed, so I wrote a node.js script to automatically upload the resources. Node.js is not the only option. We can choose other ways such as C# or Restful API to do the job. But Restful API is not convenient because we need to write some code to sign the access key.

(3) Jenkins.

Integrate the build steps and place them on GM server.

For machine assets, step 1 and 2 are integrated on Jenkins. To connect step 1 and 2, we need to generate a shell script and writes it to a file in step 1. The shell script includes some information like which machine assets should be uploaded, and node.js execution command. This is meta-programming style as I wrote some C# editor code to generate some shell script texts.

Another alternative is to generate only configurations instead of the whole script and write it to the file. If so, step 2 should read and parse the configurations before executing the script.

For all 3 types: live update, AB test, and machine assets, the generated asset bundles and meta files are all uploaded to internal FTP sites for our check when something goes wrong.