

MultiDocumentPlusUUID is a working example of an iOS 7.1 (11D5134c 11D5145e and 7.1 GM (11D167)) application with cloud-syncing UIManagedDocuments. This project is open-sourced and released under the MIT License by Don Briggs at <https://github.com/DonBriggs/MultiDocumentPlusUUID> .

See the notes below on updates to MultiDocumentPlusUUID for iOS 7.1 (11D5145e).

Purpose

MultiDocumentPlusUUID provides a limited working example of cloud-synced UIManagedDocuments. It derives from Rich Warren's MultiDocument example:
See: [Freelance Mad Science Labs - Blog - Syncing multiple Core Data documents using iCloud](#)

Credits

WWDC 2013 Video - Session 207 [What's New in Core Data and iCloud]

See: <https://developer.apple.com/wwdc/videos/index.php?id=207> .

See Drew McCormack's note:

<http://mentalfaculty.tumblr.com/post/25241910449/under-the-sheets-with-icloud-and-core-data>

Under the Sheets with iCloud and Core Data: Troubleshooting

"Unfortunately, the most apt conclusion is probably that iCloud syncing of Core Data is not really ready for prime time, at least not for any app with a complex data model. If you have a simple model, and patience, it is doable, even if very few have achieved a shipping app at this point."

Drew's observation was certainly true at the time—[June 16, 2012](#). This MultiDocumentPlusUUID effort has had a rather long and tragic history; it seems to break spectacularly with major releases of iOS. Many Bothans died... With iOS 7, we're in somewhat better shape. Perhaps we've achieved a plateau of stability in the Core Data Cloud-sync API.

See Erica Sudun's iOS 5 Developer's Cookbook:

See: <http://books.google.com/books?id=YeHQzA6UrcEC&pg=PT1173&lpg=PT1173&dq=yorn+BOOL+ios&source=bl&ots=vynUYXoVpH&sig=PCjk79J9uGt0EHHNgNF28JpQ-HA&hl=en&sa=X&ei=euGFT6K2BY-s8ATBgvyXCA&ved=0CB4Q6AEwAA#v=onepage&q=yorn%20BOOL%20ios&f=false>

See: David Trotz

See: <http://stackoverflow.com/questions/18971389/proper-use-of-icloud-fallback-stores>
<https://github.com/dtrotzjr/APManagedDocument>

This sample code includes Stav Ashuri's UIAlertView, Copyright (c) 2013. Thanks!

See: <https://github.com/stavash/UIUIAlertView>

A Broad Overview

Primarily, MultiDocumentPlusUUID uses UIManagedDocument directly. Optionally, it uses a light-weight subclass (RobustDocument) to demonstrate an approach to handling errors in Core Data.

The root view shows all the application's documents in a table view: those created locally and those discovered through NSMetadataQuery.

As with Warren's original, each document associates one-to-one with a dictionary called a record. The record acts as a helper class for the document, and avoids (postpones?) subclassing UIManagedDocument. Each cell in the root view's table view associates one-to-one with a record, and displays the status of its record's document.

The data model is simple; each document has a singleton TextEntry. Its fields include a text string and a modification date. Neglect the rest.

The application uses UUIDs to distinguish its documents. Each document has its own UUID to distinguish it from any documents with the same file name. Different devices might create documents with the same name near the same time—"Untitled" for example—but we must be able to distinguish them. In MultiDocumentPlusUUID, a document's UUID precedes its file name in both its sandbox and cloud URLs. The UUID also provides the value of `NSPersistentStoreUbiquitousContentNameKey` in the document's persistent store options.

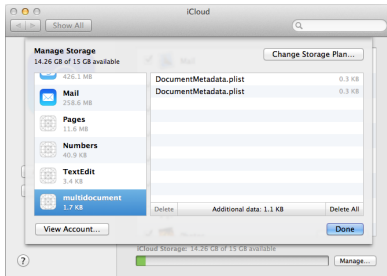
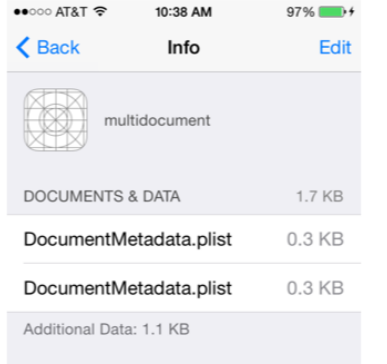
MultiDocumentPlusUUID can provide update latencies of cloud-syncing between two devices viewing the same document.

MultiDocumentPlusUUID focusses on creating and discovering documents. Generally, documents should persist between launches in the sandbox, but this limited example neglects all that. Nor is MultiDocumentPlusUUID agile over iCloud accounts. Stick to one iCloud account at a time across all devices in your tests.

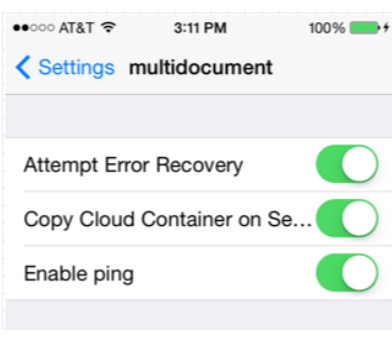
How to “Make it Go”

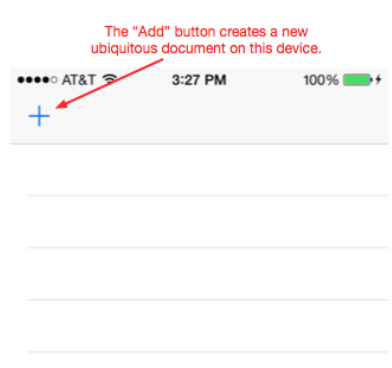
See: <https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/AddingCapabilities/AddingCapabilities.html>
[App Distribution Guide: Adding Capabilities](#)

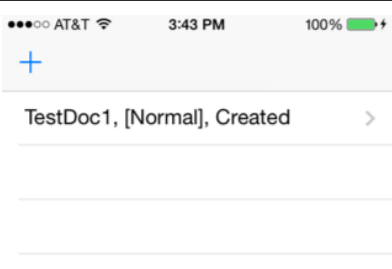
Download the project from GitHub. Add your own certificate, profile and entitlements plist to build the application with Cloud persistence. Compile the application and run it on two devices under Xcode’s debugger. Make sure to log into the same iCloud account on both devices. Typically, it seems best to clear the application’s cloud “Storage” (as in “Storage and Backup”) after each session. It’s handy to do this on the desktop, but sometimes it’s necessary to visit the Settings of each device, too. Don’t be discouraged if you find it necessary to do so.

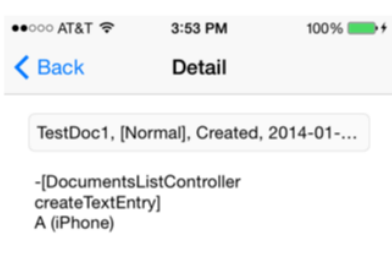
	<p>On OS X, use Preferences->iCloud</p>
	<p>On iOS, use Settings->iCloud->Storage and Backup->ManageStorage->multidocument</p>

Once the application builds and runs, use Settings.app to set a few properties on both devices. In the storyboard below, the first device is an iPhone named “A”, and the second device is an iPad named “B”.

	<p>“Attempt Error Recovery” uses RobustDocument instead of UIManagedDocument. It’s merely a starting stub for managing Core Data errors.</p> <p>“Copy Cloud Container on Segue” enables a handy tool: copy the cloud container to the sandbox. Apple suggests including such a copy in your bug reports and Technical Incidents.</p> <p>“Enable Ping” enables automatic pinging: when the detail view of a device detects a change in the document’s object model, it automatically adds a short ping message to TextEntry.text and saves the document. Just two devices can participate safely.</p>
---	--

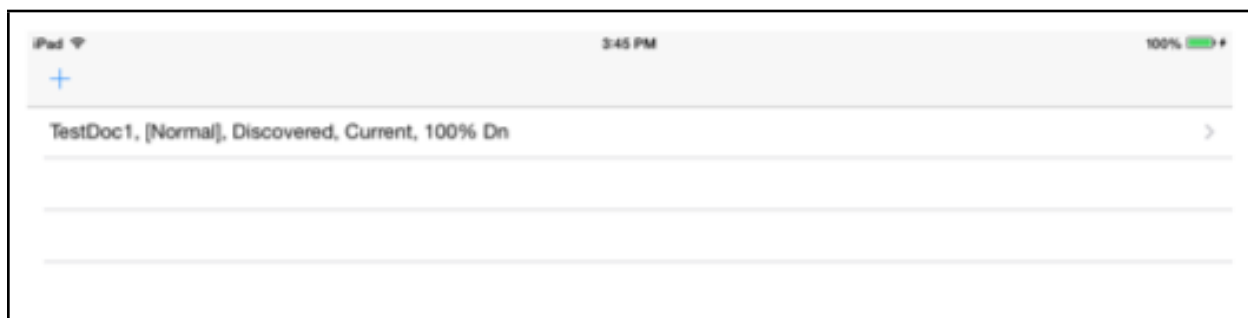
	<p>Initially, the root table views on both devices show no documents — if you’ve successfully cleared any zombie documents (below). Touch the “Add” button (+) in the upper left to create a new ubiquitous document on the first device.</p> <p>(Suddenly with OS X 10.9.2 Build 13C39, it seems much more likely that this initial screen will display zombie documents— documents previously deleted from the application’s cloud “Data and Storage.” A workaround is simply to quit and restart MultiDocumentPlusUUID from Xcode; the zombies vanish. Your mileage may vary...}</p>
--	---

	<p>Very quickly, the device’s created document appears in its root table view. The row shows the document’s status. The status includes the document’s file name, document state, and whether it was created on this device or discovered. Touch the row to segue to the detail view, and inspect the object graph.</p> <p>(In a “real” application, the user should be unaware which device created a document; this example application is for developers.)</p>
---	---

	<p>The newly created document’s object graph is quickly available for inspection in the detail view. The TextEntry’s text property shows the name and model of the device that created it.</p>
---	--



Meanwhile, the second device discovers the existence of newly created document within a few seconds. The document's status includes its download status and its "percent downloaded." (A document is discovered through an instance of `NSMetadataItem`, which has a value for `NSMetadataUbiquitousItemPercentDownloadedKey`.) But, the newly discovered document's object graph is not immediately available for inspection. Its cell is gray and disabled.



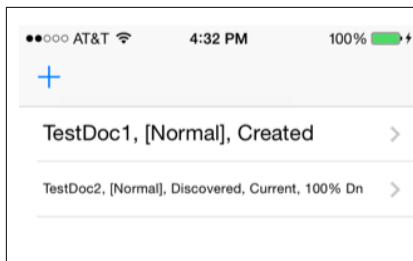
Perhaps 90 seconds later, the discovered document's object graph becomes available for inspection. Its cell becomes white and enabled: touch the row to segue to the detail view. The root view controller observes all its documents' `NSPersistentStoreDidImportUbiquitousContentChangesNotification`. Its first occurrence signals the availability of a document's object graph. (If the delay for that signal is much longer than 90 seconds, expect a Core Data error.)



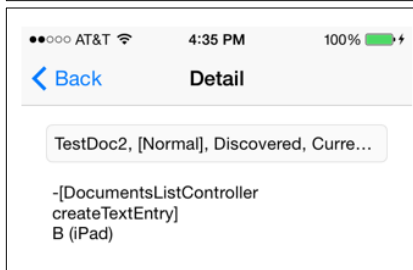
The detail view on the second device shows the same object graph as on the first device, where the document was created.



Similarly, we can create a document on the second device (TestDoc2). (A document's file name follows sequentially the file names in the table view in the creating device.)



The document created on the second device (TestDoc2) appears on the first device, in the second row.



In due time, TestDoc2's object graph becomes available on the first device.

Pinging

MultiDocumentPlusUUID includes a stunt to demonstrate sustained cloud syncing and to record the latencies of updates between two devices. With "Enable Ping" enabled on both devices, and both devices' detail views showing the same document, when either detail view detects a change to its document's object graph (NSPersistentStoreDidImportUbiquitousContentChangesNotification), it appends a short ping message programmatically.



●●○○ AT&T 4:52 PM 100%

[Back](#) Detail

TestDoc1, [Normal], Created, 2014-01-...

```
-[DocumentsListController
createTextEntry]
A (iPhone)
```

On both devices, select the same document—“Created” on one; “Discovered” on the other—and segue to both detail views.

iPad 4:57 PM 100%

[Back](#) Detail

TestDoc1, [Normal], Discovered, Current, 100% Dn, 2014-01-17 21:43:11 +0000

```
-[DocumentsListController createTextEntry]
A (iPhone) Hello, document. Having a wonderful time, wish you were ubiquitous.
```

1 2 3 4 5 6 7 8 9 0

- / : ; () \$ & @ Done

#+= undo . , ? ! ' " #+=

ABC ABC

Modify the TextEntry text on one device. (Here, we choose the iPad.)

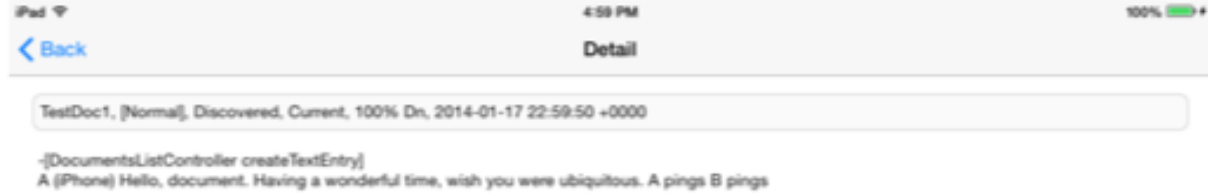
●●●● AT&T 4:59 PM 100%

[Back](#) Detail

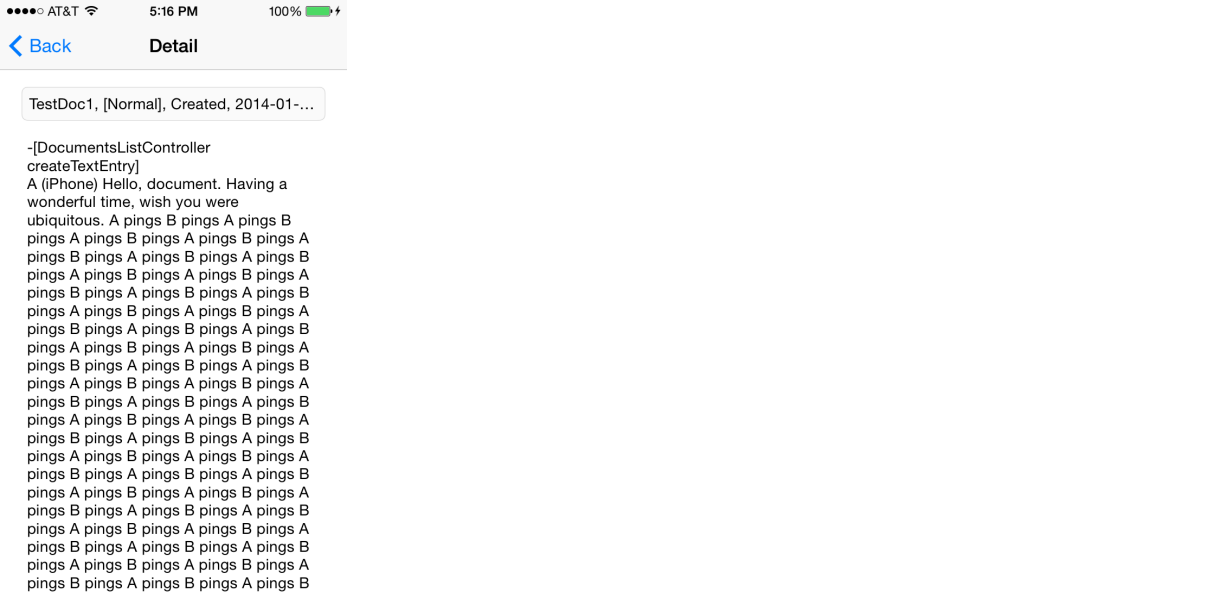
TestDoc1, [Normal], Created, 2014-01-...

```
-[DocumentsListController
createTextEntry]
A (iPhone) Hello, document. Having a
wonderful time, wish you were
ubiquitous. A pings
```

Within a few seconds, the modified text appears on the iPhone. Because “Enable Ping” is set to YES in the iPhone’s Settings,app, the receiving device quickly appends a short ping message: “A pings”.



Within a few seconds, the iPad detects the iPhone's modification and appends its own ping message, "B pings", because its Settings.app also has "Enable Ping" set to YES.



The ping process iterates. QED: a limited working example of sustained cloud-syncing of `UIManagedDocuments`. It can be hours of fun.

Latencies:

After many ping iterations, set a breakpoint in `-ping: (DetailViewController +Ping)`. In the debugger console, print the latencies:

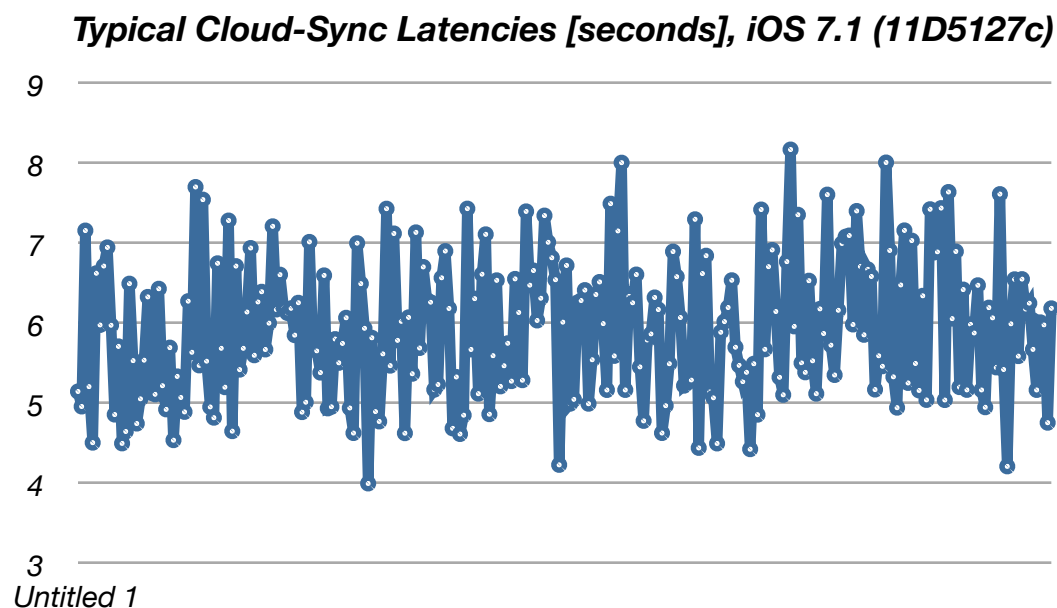
```
(lldb) po [self reportLatencies]
2014-01-17 14:17:18.623 multidocument[221:60b] 22.40558397769928
5.156464993953705
4.969072997570038
7.16798198223114
. . .
5.988353967666626
4.766466975212097
6.195563971996307
(lldb)
```

Alternatively, extract the latencies by using Xcode's Organizer window. Download the application's sandbox to the Desktop (or equivalent). Use Finder's "Show Package Contents" option, then navigate to

"AppData/Library/Preferences/com.<yourCompany>.multidocument.plist".

As of 11D5145e and this version (2014 Feb 07), copy the array named "iCloud Sync Latencies". Paste it into a new TextEdit document and remove the plist markup, and fuss with the white space. The result should be suitable to paste into a column in a Numbers document.

Here's a graph of those latencies:



Note that the latencies above came from just two devices interoperating in a very simple environment, syncing one very simple document. Your mileage may vary...

Updates for iOS 7.1 (11D5145e)

A few days ago (2014 Feb 04) I upgraded my devices to iOS 7.1 (11D5145e), and installed Xcode Version 5.1 (5B103i) aka Xcode51-Beta5.

Two things broke:

- [1] the “Storyboard/UITableView/UITableViewCell” implementation; and
- [2] how to determine when a discovered document’s object graph becomes available.

The fixes:

- [1] Xcode51-Beta5 discovered the Storyboard snafu and warned about it; it’s just pickier. I just dropped the DocumentCell class and reverted to the default UITableViewCell.

- [2] Suddenly, the previous version posted to GitHub just wouldn’t permit the user to inspect a discovered document; its row/cell remained grey (userInteractionEnabled=NO). I had relied on

NSPersistentStoreDidImportUbiquitousContentChangesNotification to signal when a document’s object graph became available. After several hours testing, I discovered that a useful signal is now:

```
NSConcreteNotification 0x165ea720 {name =
com.apple.coredata.ubiquity.importer.didfinishimport;
object = <NSPersistentStoreCoordinator: 0x16695b40>;
userInfo = {
    deleted = "{(\n)}";
    inserted = "{(\n  0x16586180 <x-coredata://E24729CE-4C32-4961-
A228-005264DCA31B/ModelVersion/p2>,\n  0x165de350 <x-coredata://
E24729CE-4C32-4961-A228-005264DCA31B/TextEntry/p2>\n)}";
    updated = "{(\n)}";
}}
```

Note that the “inserted” objects comprise the document’s initial object graph.

However it seems odd that the observer block for the notification named
NSPersistentStoreDidImportUbiquitousContentChangesNotification
catches a notification with the name

“com.apple.coredata.ubiquity.importer.didfinishimport”.

Since its arrival signals that a discovered document’s object graph has become available, I rather expected some documentation from Apple on the matter, but I’ve not seen it.

I’ve adopted a workaround.

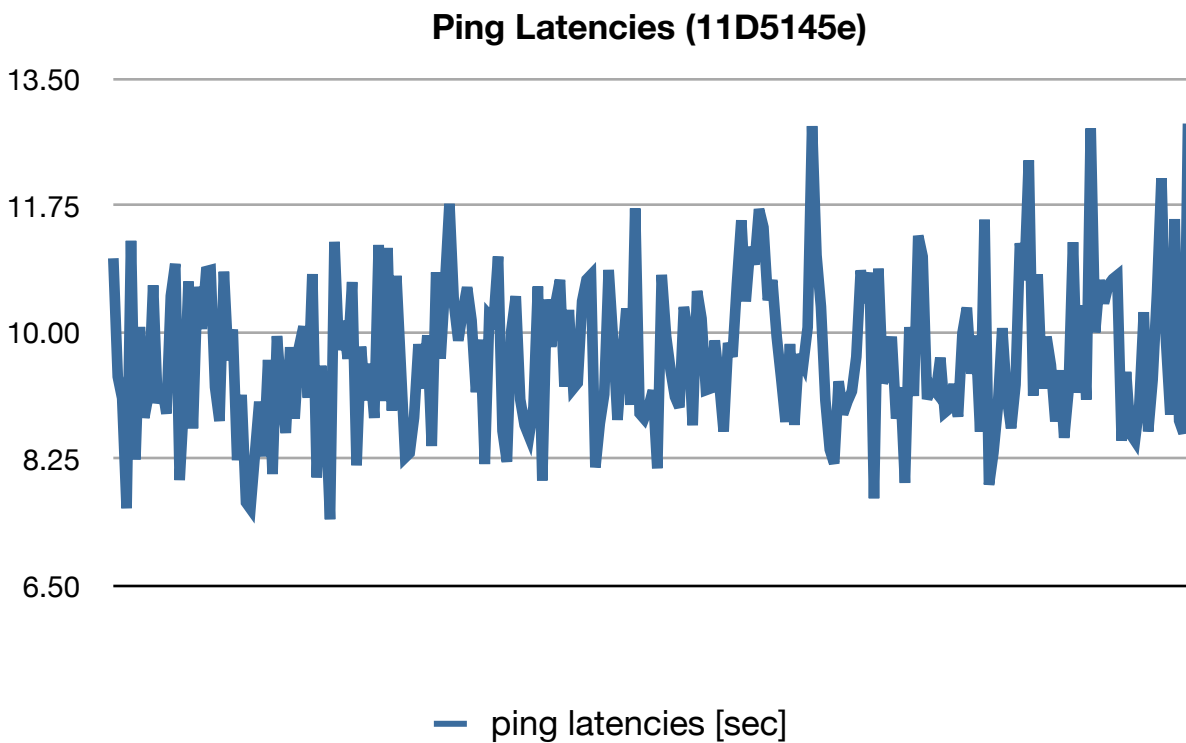
See:

- [DocumentsListController+Making receivedNotification:document:]
- and
- [NSDictionary+NPAssisting isDocumentViewable].

This workaround seems not at all future proof. I gather we've NOT achieved a long-hoped-for plateau of stability in the Core Data Cloud-sync API.

Another effect of the upgrade to 11D5145e is that the latency to access a newly created document is much longer now—perhaps 10 to 20 seconds.

Ping Latencies also seem a little longer in 11D5145e:



As always, your mileage may vary...