System Configuration Programming Guidelines

Networking

2006-02-07

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to System Configuration Programming Guidelines

This document describes the architecture of the system configuration services in Mac OS X and the APIs in the System Configuration framework. It describes how users and applications use the configurable network preferences and how applications can use the framework APIs to accomplish various tasks.

What Is the System Configuration Framework?

The System Configuration framework provides powerful, flexible support for establishing and maintaining access to configurable network and system resources. It offers your application the ability to determine, set, and maintain configuration settings and to detect and respond dynamically to changes in that information.

The framework supports a wide range of configuration management, including high-level access to network services. Although the bulk of the System Configuration APIs were available in Mac OS X version 10.1, later versions of Mac OS X have included some changes and additions. This document focuses on the System Configuration APIs available in Mac OS X version 10.3.

Who Should Read This Document?

The audience for this document comprises two main groups:

- Developers of network configuration or dialer software.
- Developers of applications that need to request and use network connections.

If you are developing an application that defines network services, initiates IP networking or creates a PPP connection, you should concentrate on the network preferences and configuration APIs. If you're developing an application that needs to know if a remote host is reachable or initiate a PPP connection, you should focus on the reachability and connection APIs.

Organization of This Document

This document is divided into four chapters:

- "System Configuration Goals and Architecture" (page 9) describes the System Configuration framework in its entirety, paying special attention to the interaction of the network preferences, configuration agents, and APIs.
- "Components of the System Configuration Framework" (page 17) describes the individual System Configuration framework components in greater detail. In this chapter, you learn about how the components work together and how the configuration agents do their work.

- "The System Configuration Schema" (page 23) describes the structure of network preferences and introduces the System Configuration schema that defines it. This chapter also describes how the configuration agents use the schema's keys and values and when you might need to use them.
- "Determining Reachability and Getting Connected" (page 47) introduces the reachability and connectivity APIs and describes how applications can use them.

The chapters you read depend on the goals of your application. All developers new to the System Configuration framework should read "System Configuration Goals and Architecture" (page 9). Then, if you're developing an ISP or dialer application that needs to manipulate network configurations, you should read "Components of the System Configuration Framework" (page 17) and "The System Configuration Schema" (page 23). If, on the other hand, you're developing an application that initiates PPP connections or determines if a remote host is reachable, you can skip to the last chapter, "Determining Reachability and Getting Connected" (page 47).

See Also

Apple provides comprehensive API reference documentation of the System Configuration framework. On the web, see *System Configuration Framework Reference*.

In addition, Apple provides several code samples that illustrate various network configuration and connection tasks. These samples are available on the web at http://developer.apple.com/samplecode/Sample_Code/Networking.htm.

When you install the Developer Package, you get developer documentation as well as tools and sample code. The System Configuration API reference documentation is available at `/Developer/ADC Reference Library/documentation/Networking/Reference`.

System Configuration Goals and Architecture

This chapter provides an overview of the System Configuration framework goals and architecture. You should read this chapter if you're new to Mac OS X network configuration or if you need to know which APIs and services in the System Configuration framework your application should use. In particular, if your application needs to determine the reachability of a remote host or request a PPP-based connection, you might choose to skim this chapter for context and then read "Determining Reachability and Getting Connected" (page 47).

Goals of the System Configuration Framework

The Mac OS X System Configuration framework provides the foundation for network configuration on Mac OS X. The dual goals of the framework are:

- To provide dynamic network configuration that supports seamless user mobility
- To support applications that need to create, modify, or access network services. This includes applications that create or manipulate network configurations and applications that need to determine remote-host reachability and make connections

The first goal of the System Configuration framework is user-oriented. In the System Preferences application's Network preferences panel (which is a client of the System Configuration framework), a user can set up multiple network configurations. Each of these configurations can describe a different networking environment. This means that a user can enter a few values in Network preferences and her system automatically detects, configures, and starts the appropriate network service. The user can impose a preferred order on the services and the system automatically switches between them, according to which network interfaces are currently available, without requiring a restart. For an example of how this works, see "An Example of Mobility" (page 10).

The second goal of the System Configuration framework is to provide network configuration and access services to a wide range of applications. At one end of the range is a network-configuration application such as one an Internet service provider (ISP) might offer to define new services and provide dial-up support. Such an application needs to access System Configuration framework components at a low level to provide network services.

At the other end of the range are what can be termed network-aware applications. For the most part, a network-aware application simply needs to use existing network services, not define new ones. Such an application might need to determine if a remote host is reachable or request a network connection so it can provide content to its users.

The System Configuration framework supports applications throughout this range by providing:

- Access to current network configuration information
- APIs that allow applications to determine the reachability of remote hosts and start PPP-based connections
- Notification of changes to network status and configurations and to system state

- A flexible schema that allows definition of and access to stored preferences and the current network configuration

The System Configuration framework offers a level of abstraction that allows your application to manage a wide range of configuration tasks. To achieve this, the framework takes advantage of Core Foundation run loop technology and property list types.

In addition, the source code for the System Configuration framework is available in Darwin as part of Apple's Open Source initiative, so you can see exactly how things work. In Darwin, the source code for the System Configuration framework and related entities is divided among three projects:

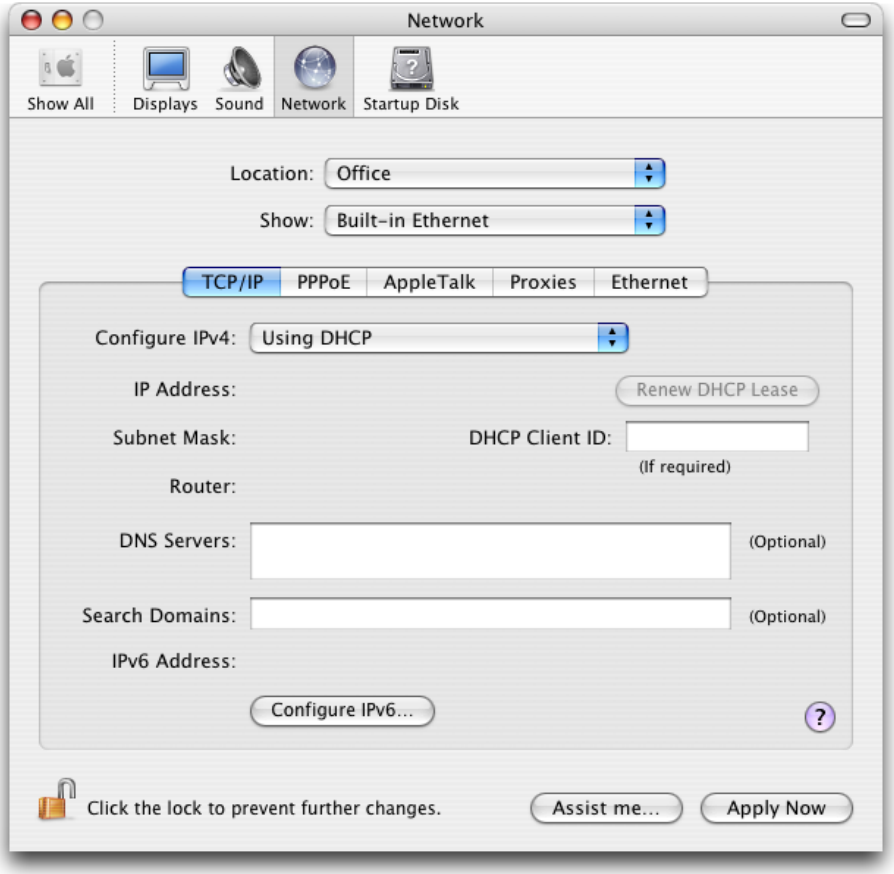
- `configd`. This project includes `configd` (the network configuration daemon) and the System Configuration framework itself.
- `configd_plugins`. This project contains various built-in plug-ins, such as the kernel event monitor and the IP monitor.
- `bootp`. This project includes BootP and DHCP, which inform the IPv4 and IPv6 configuration agents.

An Example of Mobility

The System Configuration framework allows users to set up multiple network interfaces in various combinations and supports dynamic network reconfiguration without requiring user intervention. To see how this works, consider Maria, a PowerBook-toting executive on her way to deliver a presentation at a remote office.

Because Maria uses her PowerBook in different network environments, she uses Network preferences to establish a collection of network configurations for each environment. Each collection of network configurations is called a location.Figure 1-1 (page 11) shows a location named Office in Maria's Network preferences.

Figure 1-1 A location in Network preferences

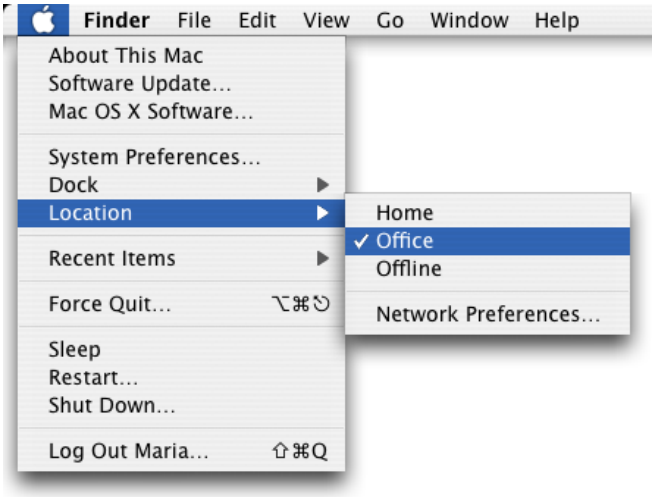


Maria establishes three locations:

- Office. The Office location sets the built-in Ethernet interface to be the primary interface. If the built-in Ethernet interface isn't available, the system uses the AirPort interface instead.
- Offline. This location disables all network interfaces so that no connection attempts are made.
- Home. This location sets an Ethernet-based DSL modem to be the primary interface. If the DSL modem isn't available, the system uses a 56K dial-up modem instead.

In her office, Maria plugs in the Ethernet cable and selects the Office location from the Apple menu, as shown in Figure 1-2 (page 12).

Figure 1-2 A location in the Apple menu



To attend a meeting, Maria unplugs the Ethernet cable and enters the conference room. When her PowerBook wakes up in the conference room, it determines that the cabled LAN is not active and automatically switches to the corporate AirPort network.

In the taxi to the airport, with no network services available, Maria writes her meeting notes. While waiting for her flight, she wakes the PowerBook and uses the wireless network available at the airport to send her presentation slides ahead to the remote office for finishing touches. To this point, Maria has not adjusted her network preferences, but she's had no problems using available network services, even across sleep-wake cycles.

As she boards the airplane, Maria uses the Apple menu to select the Offline location which shuts off all networking (as required during air travel). During the flight, she works on the introduction to her presentation.

When she arrives at the remote office, Maria again selects the Office location and gives her presentation. She uses the newly polished slides she retrieves over AirPort from the local server.

When she returns home, Maria plugs in the Ethernet and phone cables and selects the Home location from the Apple menu. Now she can work on last-minute business whether her DSL connection is active today or not.

Because of the System Configuration framework's dynamic network reconfiguration support, Maria never had to restart her PowerBook to take advantage of changing network environments.

System Configuration Architecture

The System Configuration framework comprises many components that work together to support configurable network resources. This section introduces these components and defines several terms the System Configuration framework (and this document) uses to describe a user's network configuration.

For developers of network-aware applications, this section provides enough background information to get started using the reachability and connection APIs. For an overview of all the APIs the System Configuration framework offers, read "System Configuration APIs" (page 15). Then, you may choose to skip ahead to "Determining Reachability and Getting Connected" (page 47) for specific information on how to use the reachability and connection APIs.

A developer of network-configuration applications, however, needs more in-depth information about the individual components of the System Configuration framework. To learn more about these components, you should read "Components of the System Configuration Framework" (page 17). In addition, if you're developing an application that defines new locations or services, be sure to read "The System Configuration Schema" (page 23).

System Configuration Terms

Before you read any further, be sure you're familiar with the System Configuration terms this document uses:

- **Location**. A collection of network configurations a user creates in Network preferences to describe a specific environment, such as the home environment.

  Note: Following the lead of the System Preferences application, this document uses "location" when describing a collection of network configurations in a user-interface context. To describe a collection of network configurations in a developer context, this document uses the term "set" (defined next).

- **Set**. The complete configuration for a single location. The configuration typically includes IP and interface configuration information, as well as system-wide information.
- **Network service**. A collection of network entities for a single network interface or network connection. Together, these entities contain all the information required to make the service active.
- **Network entity**. A collection of properties that are specific to some protocol or area of interest, such as PPP or DNS.
- **Interface**. The physical connection, such as Ethernet.

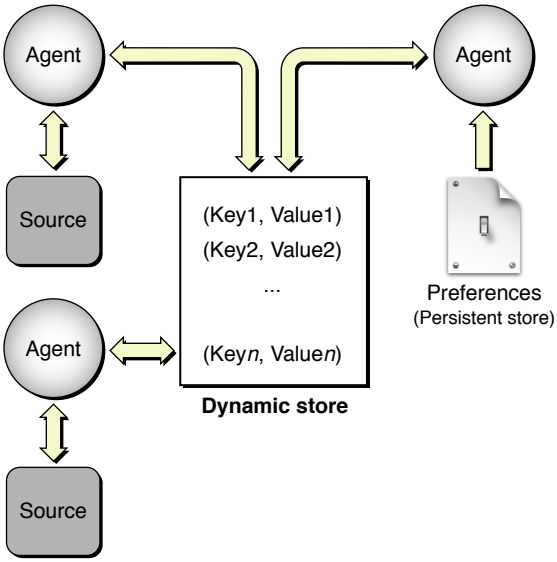
Interaction of System Configuration Components

The System Configuration framework comprises several components that work together to provide powerful, flexible support for establishing and maintaining access to configurable network resources. These components are:

- A persistent storage structure for network configuration preferences and selected system information
- A dynamic storage structure for current network state information
- Configuration agents that manage the data in the two storage areas, handle low-level configuration events, and provide notification services
- A comprehensive and flexible schema that defines both the data types in the persistent and dynamic stores and their hierarchical structure
- A full range of APIs that provide access to the current network state information and furnish notifications, handle reachability and connectivity, and support the definition of new sets and services. For an overview of the APIs the System Configuration framework provides, see "System Configuration APIs" (page 15).

Figure 1-3 (page 14) shows the first three of these components (the schema and the APIs are not shown) and how they interact.

Figure 1-3 Architecture and interaction of System Configuration framework components



As shown in Figure 1-3 (page 14), the dynamic store is the conceptual heart of the System Configuration architecture. The dynamic store:

- Contains a copy of the configuration information for the currently active location
- Contains the current network state information (and also some system state information)
- Provides input to the configuration agents
- Receives up-to-date status information from the configuration agents
- Supports the notification process

The dynamic store is kept up-to-date by various configuration agents and is recreated after every system restart. "The Dynamic Store" (page 19) describes the dynamic store in more detail.

The persistent store houses:

- The user's network preferences (entered in Network preferences)
- Some system values, such as the computer name and selected power management information

As its name implies, the persistent store persists across reboots and is only changed by explicit actions of the user, the system, or network-configuration applications. The persistent store is described in more detail in "The Persistent Store" (page 17).

Figure 1-3 (page 14) shows three configuration agents. In a running Mac OS X system, there are several more configuration agents (some of which are internal implementation details that are not discussed in this document). Each configuration agent is responsible for a well-defined area of configuration management, such as IPv4 or PPP. In general, an agent gets preferences and status information from the dynamic store

and combines it with information the agent receives from various system and network events. The agent publishes the merged information and the results of any actions it takes back into the dynamic store. See "Configuration Agents" (page 20) for more information about individual configuration agents.

Although it is not shown in Figure 1-3 (page 14), the schema serves as the common language that allows the various components of the System Configuration framework to communicate. The information in both the dynamic and persistent stores is held in key-value pairs. The configuration agents use the key-value pairs to access configuration information and to update the dynamic store. Applications can use the key-value pairs to define new services and watch for changes in network state or configuration. The schema defines these keys and values and the hierarchy that binds them together to describe specific services. The schema is described in more detail in "The Schema: Hierarchy and Definitions" (page 19). For information on how to use the schema to define new sets or services, see "The System Configuration Schema" (page 23).

The System Configuration framework also contains a full complement of APIs. These APIs include low-level functions an ISP application might use to provide new services and higher-level functions any application can use to connect to a remote host. It is important to remember that if your application is primarily concerned with the reachability of a remote host and subsequent connection to it, knowledge of the System Configuration framework architecture is not essential. Instead, you should focus on the reachability and connection APIs discussed in "Determining Reachability and Getting Connected" (page 47).

The architecture of the System Configuration framework lends itself to other types of dynamic state management, as well. The structure and flexibility of the dynamic store and the interaction of the configuration agents are well suited to the communication of system information, such as power management status. Although it is possible for third-party developers to use the dynamic store structure for purposes other than network configuration, this is not covered in this document. For help with such a project, you should contact Apple Worldwide Developer Relations directly.

System Configuration APIs

The System Configuration framework provides APIs to support a wide range of network applications, from network-aware applications that need to connect to a remote host to ISP and dialer applications.

It's useful to view the System Configuration APIs as divided into two groups:

- High-level reachability and connection APIs
- Low-level configuration APIs

Applications that need to find out if a remote host is reachable or establish a PPP connection use the high-level reachability and connection APIs. These APIs combine the power of the System Configuration architecture with the convenience of high-level functions. For more information on these APIs, see "Determining Reachability and Getting Connected" (page 47).

Applications that need to create or duplicate sets, or activate or deactivate services have a more complicated task. They must use the low-level configuration APIs. In addition, to develop these applications you must understand and use the System Configuration schema to interpret and build dictionaries that describe the new sets and services.

It's also important to realize that to modify network preferences (in other words, to change the persistent store), your application must acquire root privileges. This is not a trivial task; for more information, you can read *Authorization Services Programming Guide* and review the code samples `AuthSample` and `MoreAuthSample` available at http://developer.apple.com/samplecode/Security/idxAuthorization-date.html.

Currently, the low-level configuration APIs are very basic and somewhat difficult to use. In fact, to perform common operations, such as creating a new set, you must combine the System Configuration APIs with I/O Kit access. In future versions of Mac OS X, the System Configuration framework may provide higher-level APIs to perform such network-configuration tasks.

Until then, however, if you're developing an application that needs to create new sets or activate or deactivate services, you should examine the *MoreSCF* sample code. Apple's Developer Technical Support created this sample to provide a library of high-level functions that make performing these tasks comparatively easy. In particular, the `MoreSCF` sample provides you with some help using the complex System Configuration schema to build set and service dictionaries. Following the lead of this sample, combined with perusal of the System Configuration API reference (available in the Networking Reference Library), should provide a foundation to get you started. In addition, be sure to read "The System Configuration Schema" (page 23) to gain an understanding of the keys and values you may have to use.

System Configuration Utilities

You can use the following applications to view and, in some ways, change system configuration information:

- **scutil**. Provides command-line access to the dynamic store.
- **scselect**. Allows you to view and change the current location.
- **Property List Editor**. Provides a user-friendly way to view the persistent store.

You can use the `scutil` command-line utility to view the dynamic store, monitor notifications, and create and modify dictionaries for testing. Using the Terminal application (located at `/Applications/Utilities/Terminal`), type `scutil` at the prompt. For a list of commands, type 'help' at the `scutil` prompt.

The `scselect` tool is a setuid tool (a tool that has its setuid bit set to allow it to perform privileged operations) that the Apple menu launches to allow a non-privileged user to change locations. `scselect` is not an API. If your application needs to change locations, you should use the preferences APIs in the System Configuration framework.

For a brief description of Property List Editor, see "The Persistent Store" (page 17).

Components of the System Configuration Framework

If you're developing a network-configuration application, you need to know how the components of the System Configuration framework work together and how your application interacts with them. This chapter describes each component in greater detail.

If you're developing a network-aware application, you do not need in-depth information on the dynamic and persistent stores, the schema, or the configuration agents. Instead, you should concentrate on the reachability and connection APIs described in "Determining Reachability and Getting Connected" (page 47).

The Persistent Store

The persistent store contains the network preferences set by the user and by applications that configure network services. It is a hierarchically structured database that holds configuration information for all locations, services, and interfaces defined in the system, whether or not they are currently active. The information is kept in a large dictionary of keys (CFString types) and values (CFPropertyList types, typically CFDictionary types). The System Configuration schema (described in "The Schema: Hierarchy and Definitions" (page 19)) dictates the precise combinations of key-value pairs required to define each service and entity. The dictionary is maintained as an XML file which, in Mac OS X version 10.3 and above, currently resides in the default location `/Library/Preferences/SystemConfiguration/preferences.plist`. (The default location in earlier versions of Mac OS X is `/var/db/SystemConfiguration/preferences.xml`.)

Important: The location, type, and name of the file that represents the persistent store is private to the System Configuration framework. This document describes it to enhance your understanding of the persistent store, but under no circumstances should your application rely on its location or directly access it in any way.

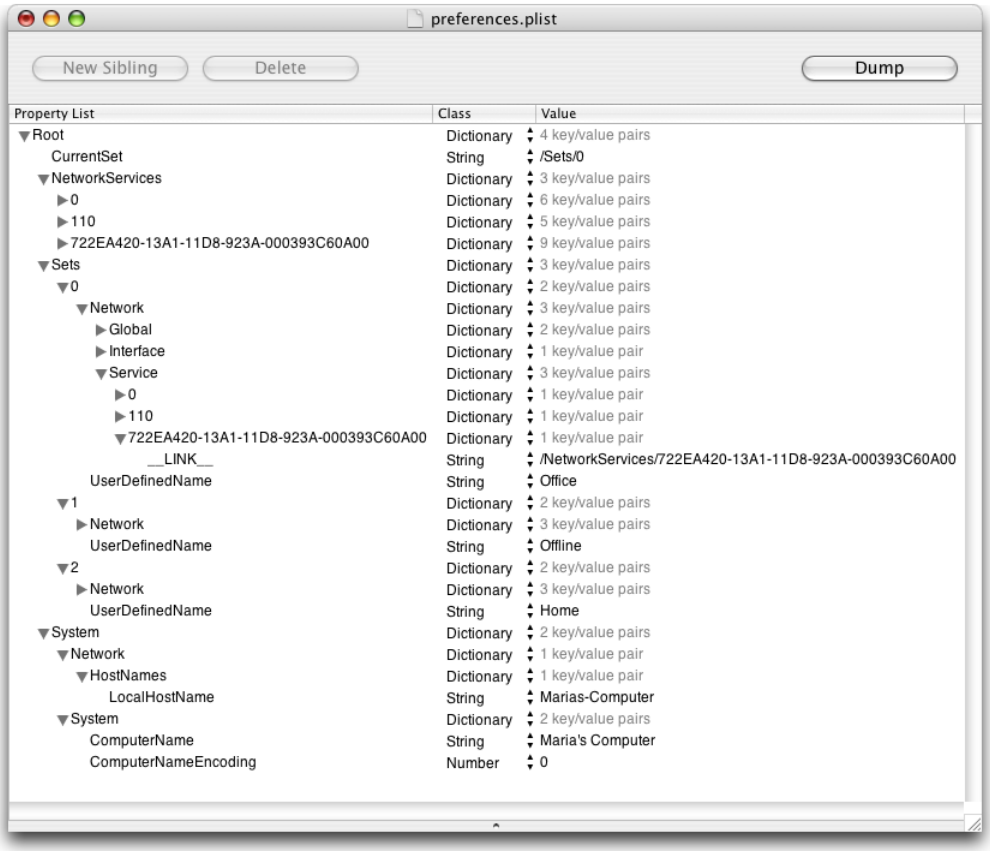
This section presents an overview of the persistent store, focusing on the top-level preferences. If you're developing a network-configuration application, you need to know how to define your service with specific key-value pairs. For more information on the precise layout of the preferences, see "The System Configuration Schema" (page 23).

Currently, the persistent store contains four top-level preferences:

- **Sets**. A CFDictionary object that lists all locations currently configured on the system, such as Office or Home. There is one set for each location. (Recall that "location" refers to the user-visible configuration of a network environment.)
- **CurrentSet**. A CFString object that contains the identity of the currently active location (a member of the Sets dictionary).
- **NetworkServices**. A CFDictionary object that contains the complete list of network services defined for all sets. Each service contains the protocol entities defined for that service.
- **System**. A CFDictionary object that contains configuration information that is system-specific rather than location-specific, such as the computer name.

Together, these four preferences describe the configurations of all locations and network services on the system. If you install the Developer Tools package, you can use the Property List Editor application (located in `/Developer/Applications/Utilities/Property List Editor`) to examine the persistent store on your computer. For example, Figure 2-1 (page 18) shows a portion of the persistent store from the computer of the fictional executive Maria (introduced in "An Example of Mobility" (page 10)), as displayed by Property List Editor:

Figure 2-1 Partial listing of a persistent store



As you can see in Figure 2-1 (page 18), the top-level preference CurrentSet contains a string that refers to a member of the Sets dictionary (in this example, the currently active location is Office).

The Sets dictionary contains a subdictionary representing the set associated with each location Maria has defined on her system. In Figure 2-1 (page 18), you can see dictionaries for the locations Office, Offline, and Home.

The NetworkServices dictionary contains a subdictionary for each network service defined on Maria's computer. Notice that in Figure 2-1 (page 18), the NetworkServices dictionary contains only the dictionaries representing the three services listed in the Office location; services listed in the other locations are not shown. Each service subdictionary in the NetworkServices dictionary is identified by a unique ID string. These ID strings have no inherent meaning and are not associated with the user-visible name for the service or location. In Mac OS X version 10.3 and above, the ID strings often consist of a GUID, or globally unique ID.

Each network service subdictionary contains the configuration information for all the entities associated with that service, such as PPP or AppleTalk. The system-wide information in the System dictionary consists of the computer name (“Maria’s Computer”) and the local host name (“Marias-Computer”).

The Dynamic Store

In a running Mac OS X system, the dynamic store contains a snapshot of the current network state. It also holds a copy of the preferences that define the currently active configuration. Because it contains the current configuration, as opposed to all configurations the user has defined, this part of the dynamic store is a subset of the persistent store. As such, the dynamic store’s hierarchical structure and combinations of key-value pairs are also determined by the System Configuration schema.

Various configuration agents (introduced in “Configuration Agents” (page 20)) keep the dynamic store up to date. To do this, they use the information in some keys to decide what to do and they publish the results of their actions in other keys. This interaction between the configuration agents and the dynamic store is an ongoing process and is not limited to the initial configuration of network services that occurs when you turn on your computer. Instead, the configuration agents update the dynamic store whenever system or network events affect the current system state. The ability of the dynamic store to reflect the current system state is at the heart of Mac OS X network mobility.

In addition to holding the current network state, the dynamic store provides a level of abstraction between low-level networking and system events and the applications that need to know about them. For example, when you unplug your Ethernet cable a configuration agent gets this information from a low-level kernel event and updates the value of the appropriate dynamic store key. Through the dynamic store, an application can get this information without having to monitor low-level system events. This is because the System Configuration framework provides notification services that allow an application to register interest in specific keys. When a key’s value changes, an interested application is notified that a change occurred. It is then the application’s responsibility to inspect the value of the key.

Although you can register interest in any key, it’s best to be very selective about the notifications you choose to receive. For one thing, to be sure you’re watching the correct key and to interpret the information you get from it, you need detailed knowledge of the complex System Configuration schema. In many cases, you can use System Configuration APIs to receive notifications of specific events without resorting to watching individual keys. For information on some of these APIs, see “System Configuration APIs” (page 15) and “Determining Reachability and Getting Connected” (page 47).

The Schema: Hierarchy and Definitions

The System Configuration schema describes the complex hierarchical layout of the persistent and dynamic stores. In addition to imposing the overall structure, the schema also defines the exact combinations of key-value pairs that describe all services available on the system. All configuration agents must fully understand the part of the schema that defines their area of interest to be able to read and correctly update the dynamic store.

Although the schema is not private system API, it is very low level. As much as possible, an application should avoid depending directly on the schema and employ higher-level interfaces instead. For example, a network-aware application can use the notification services in the reachability and connection APIs instead of requesting notifications on specific keys the schema defines. However, for some applications this is not

an option. A network-configuration application, for example, must combine the correct key-value pairs to define new locations and services. Developers of such applications should read "The System Configuration Schema" (page 23) for more information on how the schema defines specific services.

The `SCSchemaDefinitions.h` file in the System Configuration framework defines the keys and values the schema uses, but does not indicate how they're used. This section gives a brief overview of the types of keys and values you find in the `SCSchemaDefinitions.h` file. For in-depth information on the keys and values of the schema, see "The System Configuration Schema" (page 23).

The `SCSchemaDefinitions.h` file groups the keys and values into the following types:

- Generic keys. These are keys, such as `kSCPropUserDefinedName` and `kSCPropVersion`, that can be used at different levels in the persistent store. For example, `kSCPropUserDefinedName` is an appropriate key for both a service dictionary and a location dictionary.
- Preference keys. These keys define the top-level preferences in the persistent store, such as `kSCPrefCurrentSet` and `kSCPrefSystem`. (The top-level preferences are described in "The Persistent Store" (page 17).)
- Component keys. These keys define the main categories in the dynamic and persistent stores, such as `kSCCompSystem` and `kSCCompInterface`.
- Entity keys. The entity keys name network entities, such as `kSCEntNetIPv4` and `kSCEntNetDNS`.
- Property keys. These keys identify the properties for each entity, such as the IPv4 `kSCPropNetIPv4ConfigMethod` property.
- Value keys. These keys provide appropriate values for specific property keys. For example, `kSCValNetIPv4ConfigMethodBOOTP` is a possible value for the IPv4 `kSCPropNetIPv4ConfigMethod` property.

Configuration Agents

The System Configuration framework communicates with a system-level daemon, `configd`, to manage network configuration. When you turn on your computer, `configd` runs early in the boot process to configure the network. To keep the network state data current, `configd` initializes the dynamic store and loads the configuration agents as bundles (or plug-ins). These agents run within the `configd` memory space, as part of its process. Each agent is responsible for a well-defined aspect of configuration management, such as IPv4 or PPP. An agent monitors low-level kernel events, relevant configuration sources, and the status reported by other configuration agents, to configure its area of interest and update the dynamic store.

To communicate with the dynamic store, the configuration agents use the keys and values defined in the `SCSchemaDefinitions.h` file and follow the hierarchy defined by the System Configuration schema. Each agent understands a well-defined subset of key-value pairs that relates to its area of interest. The agent uses some keys to obtain configuration information and others to publish the results of events it detects and actions it takes.

In Mac OS X, `configd` loads several configuration agents, some of which are of internal interest only. Your application might be aware of the following configuration agents:

- Preferences monitor. Populates the dynamic store with the preferences associated with the currently active configuration set (specified in the user's CurrentSet preference).

- Kernel event monitor. Maintains information about the network interfaces defined in the system and monitors low-level kernel events.
- IPv4 configuration agent. Configures Ethernet-type devices for IP networking.
- IPv6 configuration agent. Configures Ethernet-type devices, FireWire devices, and 6to4 interfaces for IP networking.
- IP monitor. Selects the primary network service (the service associated with the default route and default DNS for the system).
- PPP controller. Configures PPP interfaces for IP networking.

For more information on these configuration agents and the specific keys and values they use, see "Behavior of the Configuration Agents" (page 35).

The System Configuration Schema

The System Configuration schema defines the layout of network configuration information in the persistent store. This chapter describes this layout and some of the specific combinations of key-value pairs that define sets and services. It then describes which key-value pairs the configuration agents use to manage their areas of interest. Finally, this chapter describes how to use your knowledge of the schema to provide preferences that define a service. You should read this chapter if your application defines sets or services. Additionally, if your application needs detailed information about the status of a PPP connection, you should read this chapter to find out how to interpret the information you receive.

Most network-aware applications do not need to access network preferences or the dynamic store to perform their tasks. If your application requests network connections or finds out if a remote server is reachable, you might choose to skip ahead to "Determining Reachability and Getting Connected" (page 47).

Layout of the Persistent Store

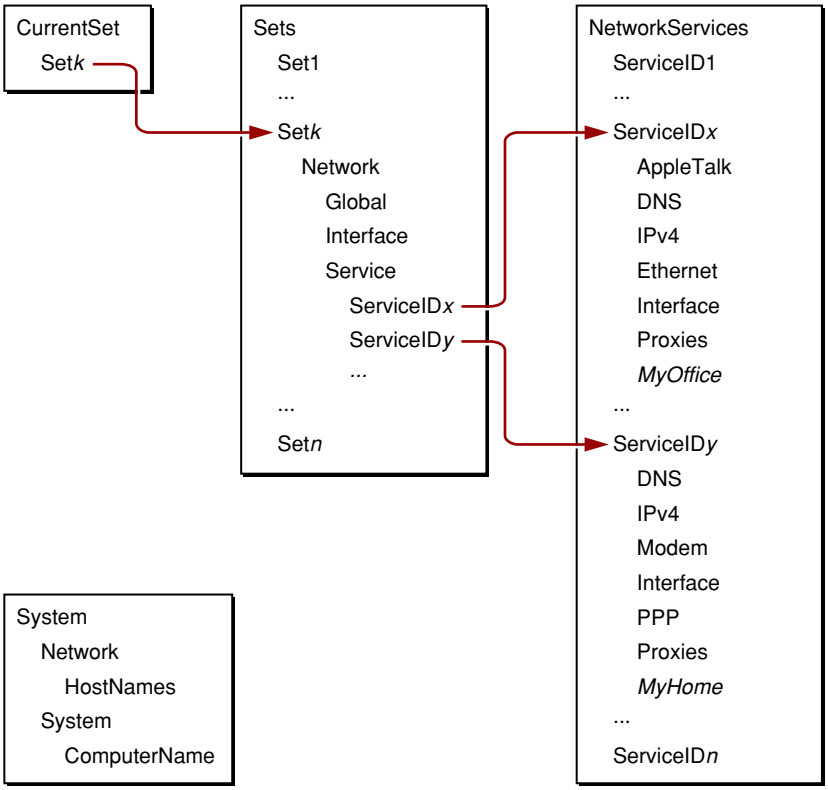
The hierarchical placement of keys and values in the persistent store is dictated by the System Configuration schema. This section describes the hierarchy of key-value pairs and identifies some of the specific combinations of preferences that define individual services.

"The Persistent Store" (page 17) introduces the four top-level preferences in the persistent store:

- Sets
- CurrentSet
- NetworkServices
- System

The system uses the information in these four preferences to configure network services. Figure 3-1 (page 24) shows how these preferences are related.

Figure 3-1 Relationship among top-level preferences in the persistent store



As you can see in Figure 3-1 (page 24), the System preference is separate from the other three preferences. This is because the System dictionary contains global, system-wide information, such as the computer name. This information is applicable to the machine as a whole, regardless of the current network configuration. A network-configuration application has no need to add values to the System preference.

The CurrentSet preference always contains the internal ID that represents the currently active set in the Sets preference (in Figure 3-1 (page 24), this is Set*k*). When a user selects a different location in Network preferences or from the Apple menu, the preferences monitor notices and updates the dynamic store to reflect the change. For more information on the preferences monitor, see "Preferences Monitor " (page 35)).

The remaining two top-level preferences, Sets and NetworkServices, contain the bulk of the information the system needs to configure network services. As you can see in Figure 3-1 (page 24), a set listed in the Sets dictionary contains the internal IDs that represent individual services listed in the NetworkServices dictionary. The following sections describe the Sets and NetworkServices dictionaries in more detail.

The Sets Dictionary

The Sets dictionary contains a subdictionary for each set defined in the system. Generally, sets represent the locations created by the user in Network preferences, although network-configuration applications can also create them. For example, an ISP might provide an application that defines a new location that makes it easy for a user to connect to the ISP. To offer this new location, the application creates a new set dictionary and some number of new services. The new set dictionary contains configuration information for the location, including a property that holds the ISP's name, and references to the new services.

Each individual set dictionary within the top-level Sets dictionary contains the following properties:

- UserDefinedName
- Network

The **UserDefinedName** property is a string that holds the location name the user enters in Network preferences. An application that defines a new location would use this key to provide a suitable location name, for example, the name of the ISP. The ISP's name is then displayed in the Location menu of Network preferences and in the Apple menu.

The **Network** property is a dictionary that describes the network configuration for the set, including a list of services defined for the set. Every Network dictionary within a set should contain the following properties, all of which are dictionaries:

- Global
- Interface
- Service

As its name implies, the **Global** dictionary supplies information that is not specific to any particular service. Currently, the Global dictionary contains two member dictionaries: IPv4 and NetInfo, both of which are required.

The IPv4 dictionary should contain the ServiceOrder key. This key's value is an array that allows the user to impose an ordering on his network services in Network preferences. When multiple services are active at the same time, the system uses the ServiceOrder array to determine which service should be deemed primary. The primary service determines the default route and the DNS and proxy server configurations.

The IPv4 dictionary may also contain the PPPOverridePrimary key. This is a legacy key that allows any active PPP-based service to be treated as if it were first in the ServiceOrder array. This functionality has been replaced by allowing any individual service to include an OverridePrimary key in its own (per service) IPv4 dictionary.

The NetInfo dictionary contains NetInfo binding preferences. As the use of Directory Services becomes more widespread, however, the NetInfo dictionary will become less common.

The **Interface** dictionary holds per-interface settings. Each member of the Interface dictionary is a subdictionary identified by the BSD name of the interface, such as `en0`. Each subdictionary contains a dictionary named Ethernet, which holds the results of a manual configuration of the interface. Speed, duplex, and maximum packet size (maximum transfer unit, or MTU) values are stored here. This information is stored at the interface level to prevent individual services from attempting to configure the same interface in different ways.

Typically, however, the user accepts the default Ethernet configuration by choosing the Automatically option in the Ethernet configuration tab of Network preferences. When this is the case, the Ethernet dictionary is either not present or contains the key `__INACTIVE__`.

The **Service** dictionary contains references to the services defined for this set. Every member of the Service dictionary is itself a dictionary that contains a single member—an internal service ID that refers to a service listed in the top-level NetworkServices dictionary. Network preferences uses the information in the Service dictionary to display the services and whether or not they're enabled. The preferences monitor also uses this information as it loads the current configuration preferences into the dynamic store (for more information on this process, see "Preferences Monitor " (page 35)).

The NetworkServices Dictionary

The top-level NetworkServices dictionary contains all the network services defined for the system. Each network service is represented by a dictionary (identified by a unique key) that contains the configuration properties for the entities associated with that service. The combination of entities is determined by the type of service. For example, a modem-based service needs the Interface, PPP, Modem, and IPv4 entities, but not the AppleTalk entity, because Mac OS X does not support AppleTalk over PPP.

If you are developing an application that adds a service to a new set, you need to create a network service dictionary that describes it. This section outlines the contents of a network service dictionary.

A network service dictionary can include the following properties:

- An interface dictionary that describes the network interface underlying the service

  Note: If you want to ensure that the appropriate configuration agent processes your service, you must include an interface dictionary in your network service dictionary.

- A variable number of protocol entity dictionaries, each of which describes the configuration of a protocol entity (such as PPP or AppleTalk) that's defined for the service. (Note that some protocol entity dictionaries may be present in a network service dictionary but be empty or marked as inactive.)
- One hardware entity dictionary, depending on the type of service (the type of service is specified in the interface entity dictionary)
- A proxies dictionary that identifies which proxies are enabled for the service
- A user-visible name for the service

The Interface Dictionary

The Interface dictionary describes the transport of the service's network connection. The contents of the Interface dictionary vary according to the type of the service, but three members are required:

- DeviceName
- Hardware
- Type

The value of the **DeviceName** key supplies the BSD device name for the network connection, for example, `en0` or `en1`. For serial-like devices, the value of the DeviceName key is the base name, minus any prefix. For example, the DeviceName value for a modem with the name `cu.modem` is `modem`. Currently, the System Configuration framework does not supply API to find all ports suitable for networking. See "Programmatically Setting Preferences" (page 44) for a reference to a library of sample code that can help with this.

The value of the **Hardware** key is a property of type CFString that names the type of hardware underlying the network connection, such as Ethernet or AirPort. You can view the contents of this dictionary as a reference to the hardware entity dictionary of the same name (for more information on hardware entity dictionaries, see "Hardware Entity Dictionaries" (page 32)).

The **Type** key identifies the type of connection, which could be Ethernet, FireWire, PPP, or 6to4.

Note: If the service is PPP, the value of the Type key must be PPP and a SubType key must be added to indicate the type of PPP, such as PPPoE (PPP over Ethernet).

In addition to the required keys, the Interface dictionary usually contains the **UserDefinedName** key, which holds the user-visible name of the network port.

The Interface dictionary keys and values are summarized in Table 3-1, along with associated actions of configuration agents.

Table 3-1 Keys and values for the Interface dictionary

Key	Value	Further action required	Notes
DeviceName	BSD device name	None	
Hardware	AirPort	None	
Hardware	Ethernet	None	
Hardware	Modem	None	
Type	Ethernet	None	
Type	FireWire	None	
Type	6to4	None	
Type	PPP	Must supply a SubType key.	PPP controller configures this service.
SubType	PPPoE	Required for PPP over Ethernet services.	
SubType	PPPSerial	Required for PPP over serial, modem, Bluetooth, etc. services.	
SubType	PPTP	Required for VPN (using PPTP) services.	
SubType	L2TP	Required for VPN (using L2TP) services.	
UserDefinedName	Human readable name	None	

Protocol Entity Dictionaries

Each protocol entity dictionary contains the information the system needs to configure the protocol. For example, the ConfigMethod property in the IPv4 entity dictionary might tell the system to use DHCP to configure IPv4.

Currently, the protocol entities that can be defined for a service are:

- AppleTalk
- DNS

- IPv4
- IPv6
- PPP

For the most part, a service dictionary can contain any combination of these protocol entity dictionaries. If an interface does not support a protocol, however, the corresponding dictionary is ignored. For example, in Mac OS X version 10.3, the AppleTalk dictionary is ignored for PPP services because Mac OS X does not support AppleTalk over PPP.

The contents of some protocol entity dictionaries (such as PPP) affect the contents of other members of a network service dictionary. The following sections describe the contents of each protocol entity dictionary, including any effect those contents might have on other properties.

AppleTalk

The AppleTalk protocol entity dictionary is an optional protocol entity dictionary. If it is present, it contains the following required key:

- ConfigMethod (`kSCValNetAppleTalkConfigMethod`)

The possible values for the ConfigMethod key are:

- Node (`kSCPropNetAppleTalkConfigMethodNode`). The standard AppleTalk configuration, which is suitable for general workstations.
- Router (`kSCPropNetAppleTalkConfigMethodRouter`). Used for AppleTalk routers.
- SeedRouter (`kSCPropNetAppleTalkConfigMethodSeedRouter`). Used for AppleTalk seed routers.

Depending on which configuration method you choose, other key-value pairs may be required. For example, if you select the SeedRouter configuration method, you must also supply values for the SeedNetworkRange and SeedZones keys.

For more information on AppleTalk configuration options, you can view the man page for the `appletalk.cfg` file (the default configuration file the `appletalk` command reads) at http://developer.apple.com/documentation/Darwin/Reference/ManPages/index.html.

The key-value pairs in the AppleTalk protocol entity dictionary have no effect on the contents of other dictionaries in a network service dictionary.

DNS

The DNS protocol entity dictionary is an optional protocol entity dictionary. It is usually present in a service's network service dictionary, but is often empty. An empty DNS protocol dictionary means that the user chooses not to manually configure DNS, allowing the DNS server names and search domains to be automatically supplied by the ISP.

If you want to require the use of specific DNS servers and search domains, you can add any of the following optional keys to the DNS protocol entity dictionary:

- DomainName (`kSCPropNetDNSDomainName`)
- SearchDomains (`kSCPropNetDNSSearchDomains`)

- ServerAddresses (`kSCPropNetDNSServerAddresses`)
- SortList (`kSCPropNetDNSSortList`)

For more information on the SortList key, you can view the man page for the resolver configuration file format at http://developer.apple.com/documentation/Darwin/Reference/ManPages/index.html.

The key-value pairs in the DNS protocol entity dictionary have no effect on the contents of other dictionaries in a network service dictionary.

IPv4

The IPv4 protocol entity dictionary is an optional protocol entity dictionary. It has one required key:

- ConfigMethod (`kSCPropNetIPv4ConfigMethod`)

Table 3-2 (page 29) shows the six possible values of the ConfigMethod key. It also shows the additional keys you must supply with certain ConfigMethod values.

Table 3-2 Optional values for the IPv4 ConfigMethod key

Value	Further action required
BOOTP	None
DHCP	None
LinkLocal	None
INFORM	Provide values for Addresses, Router, SubnetMasks `Setup:` keys.
Manual	Provide values for Addresses, Router, SubnetMasks `Setup:` keys.
PPP	Can specify a manual address.

For all ConfigMethod values except PPP, the IPv4 configuration agent publishes values for the Addresses, SubnetMasks, Router, and InterfaceName `State:` keys. When the ConfigMethod is PPP, the PPP controller publishes values for the Addresses, DestAddresses, Router, and InterfaceName `State:` keys. For more information on these configuration agents, see "Behavior of the Configuration Agents" (page 35).

IPv6

The IPv6 protocol entity dictionary is an optional protocol entity dictionary. The IPv6 protocol entity dictionary has much the same composition as the IPv4 protocol entity dictionary and it, too, has a single required key:

- ConfigMethod (`kSCPropNetIPv6ConfigMethod`)

Table 3-3 (page 30) shows the four possible values of the ConfigMethod key, along with any further action required.

Table 3-3 Optional values for the IPv6 ConfigMethod key

Value	Further action required	Notes
Automatic	None	IPv6 configuration agent publishes values for Addresses, PrefixLength, Flags, and InterfaceName `State:` keys.
Manual	Provide values for Addresses, Router, and PrefixLength	IPv6 configuration agent publishes values for Addresses, PrefixLength, Flags, Router, and InterfaceName `State:` keys.
RouterAdvertisement	None	IPv6 configuration agent publishes values for PrefixLength, Flags, and InterfaceName `State:` keys.
6to4	Provide value for 6to4 Relay	IPv6 configuration agent publishes values for Addresses, PrefixLength, Flags, and InterfaceName `State:` keys.

If an IPv6 protocol entity dictionary is present in a network service dictionary, the PPP controller triggers IPv6 negotiation within PPP. It then configures the LinkLocal address if the IPv6 negotiation with the server was successful.

PPP

The PPP protocol entity dictionary is an optional protocol entity dictionary. The schema provides many keys and values for use in this dictionary to allow developers to define finely grained services. The content of the PPP protocol entity dictionary affects the contents of some of the other dictionaries in a network service, most notably, the IPv4 and Interface dictionaries. In addition, the presence of a PPP service in a set requires the presence of the PPPOverridePrimary key in the IPv4 subdictionary in the set's Global dictionary (as described in "The Sets Dictionary" (page 24)).

Mac OS X recognizes a wide range of options for PPP services, many of which are used only in very special circumstances. This document does not attempt to describe them all. Instead, this section focuses on the PPP options a user sees in Network preferences. Because you can connect to the vast majority of PPP services using these more generic options, you can create a PPP protocol entity dictionary using the keys described in this section (and their default values), and your connection should succeed. Then, you can refine your service by adding other keys defined in the `SCSchemaDefinitions.h` file.

Table 3-4 shows the keys for the user-visible PPP options, along with some possible values.

Table 3-4 Optional keys and values for the PPP protocol entity dictionary

Key	Value	Notes
DialOnDemand	0	Enables automatic dial-up. This key should be used with care to avoid unwanted dialing.
IdleReminder	0	Prompts the user to maintain the PPP connection when the seconds in the IdleReminderTimer value elapse. If the user doesn't acknowledge the onscreen dialog, PPP automatically disconnects.

Key	Value	Notes
IdleReminderTimer	1800	The number of seconds to elapse before the user is reminded to maintain the PPP connection (if IdleReminder is enabled).
DisconnectOnIdle	1	Automatically disconnects if there is no traffic for the number of seconds in the DisconnectOnIdleTimer value.
DisconnectOnIdleTimer	900	The number of idle seconds to elapse before the PPP connection automatically disconnects (if DisconnectOnIdle is enabled).
DisconnectOnLogout	1	Automatically disconnects when the user logs out or when the user switches out with fast user switching.
CommRedialEnabled	1	For modem-based connections, enables redial if busy.
CommRedialCount	1	The number of times to redial if busy (if CommRedialEnabled is enabled).
CommRedialInterval	30	The interval in seconds to wait between redialing if busy (if CommRedialEnabled is enabled).
LCPEchoEnabled	1	Enables the PPP echo protocol. This protocol allows the rapid detection of an unreported modem disconnection. Enabling this protocol generates additional traffic and may not be suitable if the user pays by traffic load instead of by connection time.
IPCPCompressionVJ	1	Enables VJ (Van Jacobson) compression on the PPP link. For PPPSerial connections, the value can be 1 (“on”) or 0 (“off”). For PPPoE, PPTP, and L2TP connections, the value should be 0 (“off”).
CommDisplayTerminalWindow	0	Enables display of a Terminal window to show login and password interaction with a server. This works only with PPPSerial connections.
VerboseLogging	0	Enables the logging of the entire PPP negotiation, which can be useful for debugging a connection failure.
UserDefinedName		Optional name for the connection. This value has no effect on the PPP protocol.
AuthName		User name for authentication purposes.
AuthPassword		Password for authentication purposes.
CommRemoteAddress		The address or name of the remote server. This can be a phone number for a PPPSerial connection, a ServiceName for a PPPoE connection, or an IP address or name for a PPTP or L2TP connection.
CommAlternateRemoteAddress		Used only for PPPSerial connections, this is an alternate address to use when the address in CommRemoteAddress is busy.

The definition of a PPP service requires a few changes in some of the other members of your network service dictionary. If you are providing a PPP service, you need to make the following changes to the network service dictionary you are building:

- The Type key in the Interface dictionary must have the value PPP. This ensures the PPP Controller agent will act on this service.
- You must add the SubType key to the Interface dictionary and give it the value that describes the type of PPP service you're providing, such as PPPoE or L2TP.
- If you're defining a PPP service meant to be used over a modem, you'll need to add a modem hardware entity dictionary to the network service dictionary.

Hardware Entity Dictionaries

Your network service dictionary must contain exactly one hardware entity dictionary, which should provide information on a per-service basis. Its name must match the value of the Hardware property of the Interface dictionary (described in "The Interface Dictionary" (page 26)). The hardware entity dictionary can be:

- Ethernet
- AirPort
- Modem

A hardware entity dictionary contains details about the network hardware underlying the service defined by the network service dictionary. Therefore, the keys and values in the dictionary vary according to the type of hardware the dictionary describes.

The **AirPort** hardware entity dictionary can contain some combination of the keys shown in Table 3-5.

Note: In versions of Mac OS X later than 10.3, the contents of the AirPort hardware entity dictionary may move to another location.

Table 3-5 Keys and values for the AirPort hardware entity dictionary

Key	Value
AllowNetCreation	0 or 1
AuthPassword	
AuthPasswordEncryption	Keychain
JoinMode	Automatic
JoinMode	Preferred
JoinMode	Recent
JoinMode	Strongest
PowerEnabled	0 or 1

Key	Value
PreferredNetwork	
SavePasswords	0 or 1

The **Modem** hardware entity dictionary supports a large number of options that allow you to specify a highly detailed setup. Table 3-6 displays some of the keys the schema defines along with some typical values.

Table 3-6 Keys and typical values for the Modem hardware entity dictionary

Key	Value	Notes
ConnectionScript	Name of file in modem scripts folder	This is the CCL (connection control language) script.
DataCompression	1	
DialMode	WaitForDialTone	Other values are IgnoreDialTone and Manual.
ErrorCorrection	1	
HoldCallWaitingAudibleAlert	0 or 1	
HoldDisconnectOnAnswer	0 or 1	
HoldEnabled	0 or 1	
HoldReminder	0 or 1	
HoldReminderTime		
PulseDial	0	
Speaker	1	
Speed		The speed the serial port is opened with to communicate with the modem.

As the developer of a network-configuration application, you supply the modem options required by your setup. When a user opens Network preferences, she can set the options that matter to her. For example, an application can set the ConnectionScript property to point to a file in the modem scripts folder at `/Library/Modem Scripts`. A user might choose different Speaker and DialMode options depending on the environment in which she is making the connection.

The Proxies Dictionary

The Proxies dictionary in each network service dictionary contains the proxy server configuration information. The System Configuration schema defines several properties that allow a user to specify how to configure proxies for the given service. For example, a user can:

- Choose which proxies to enable

- Identify the server and port associated with enabled proxies
- List network resources that should be accessed without a proxy server

A network-configuration application might deliver a custom set of default proxy information for the user to accept or modify. To do this, the application supplies a Proxies dictionary that includes specific proxy servers, ports, and enable status.

For most proxy types, the `SCSchemaDefinitions.h` file defines three keys that provide the name of the proxy, the port, and whether or not the proxy is enabled. The keys are of the form *Protocol*Proxy, *Protocol*Port, and ProtocolEnable, where *Protocol* is the name of the service to be proxied. In addition, the schema defines the ExceptionsList key which allows you to list specific hosts and domains for which the proxy settings should be bypassed. Table 3-7 lists the keys and values you can use to create a Proxies dictionary.

Table 3-7 Keys and values for the Proxies dictionary

Key	Value	Notes
ExceptionsList	Array of host and domain names	
FTPPassive	0 or 1	Controls whether or not FTP clients use passive FTP. Note that passive FTP can cause problems with older servers and some network configurations. Default value is 1 (passive FTP enabled).
FTPEnable	0 or 1	If the value of FTPEnable is 1, the FTP proxy server is specified by the value of the FTPProxy key.
FTPPort	Port number	
FTPProxy	Proxy server	
GopherEnable	0 or 1	If the value of GopherEnable is 1, the gopher proxy server is specified by the value of the GopherProxy key.
GopherPort	Port number	
GopherProxy	Proxy server	
HTTPEnable	0 or 1	If the value of HTTPEnable is 1, the HTTP proxy server is specified by the value of the HTTPProxy key.
HTTPPort	Port number	
HTTPProxy	Proxy server	
HTTPSEnable	0 or 1	If the value of HTTPSEnable is 1, the HTTPS proxy server is specified by the value of the HTTPSProxy key.
HTTPSPort	Port number	
HTTPSProxy	Proxy server	

Key	Value	Notes
RTSPEnable	0 or 1	If the value of RTSPEnable is 1, the RTSP proxy server is specified by the value of the RTSPProxy key.
RTSPPort	Port number	
RTSPProxy	Proxy server	
SOCKSEnable	0 or 1	If the value of SOCKSProxy is 1, the SOCKS proxy server is specified by the value of the SOCKSProxy key.
SOCKSPort	Port number	
SOCKSProxy	Proxy server	

Behavior of the Configuration Agents

Recall that the system-level daemon, `configd`, keeps the network state data current by initializing the dynamic store and loading the configuration agents as bundles (or plug-ins). The agents run within the `configd` memory space and are each responsible for a well-defined area of configuration management.

The configuration agents use the keys and values defined in the `SCSchemaDefinitions.h` file to communicate with the dynamic store. The agents use some keys to obtain configuration information and others to publish the results of events they detect and actions they take.

Although some configuration agents are of internal interest only, others affect the dynamic store in ways your application might need to understand. This section describes the behavior of these configuration agents and lists many of the keys and values they use.

Preferences Monitor

The preferences monitor reads the currently active configuration set specified by the user's CurrentSet preference and loads the associated preferences into the dynamic store. Loading the preferences involves a mapping process that flattens into a single level the hierarchy of nested dictionaries that make up the currently active set. The mapping replaces each nested dictionary with a new key-value pair. The new key is the path from the root of the set dictionary to the nested dictionary and the value is the nested dictionary's value, minus any additional nested dictionaries. In this way, an arbitrarily deep hierarchy of nested dictionaries can be represented as a series of top-level dictionaries, none of which contain nested dictionaries.

The mapping process also removes dictionaries that are empty or marked inactive and it resolves all references and links. It's important to note that a dictionary is considered inactive when it contains the `__INACTIVE__` key, regardless of the key's value.

As it maps the contents of the set dictionary into the dynamic store, the preferences monitor prefixes each key with the string `Setup:`. This identifies these keys as having come from the user's preferences in the persistent store. Other keys in the dynamic store begin with other strings, such as `State:`. The `State:` keys are those the other configuration agents use to hold transient network and system state information.

As an example of the mapping process the preferences monitor performs, Listing 3-1 shows a subset of a user's preferences as it appears in the persistent store.

Listing 3-1 Subset of service dictionary (`ServiceID` 100)

```
<key>100</key>
<dict>
    <key>IPv4</key>
    <dict>
        <key>ConfigMethod</key>
        <string>BootP</string>
    </dict>
    <key>DNS</key>
    </dict>
    <key>Interface</key>
    <dict>
        <key>DeviceName</key>
        <string>en0</string>
        <key>Hardware</key>
        <string>Ethernet</string>
        <key>Type</key>
        <string>Ethernet</string>
        <key>UserDefinedName</key>
        <string>Built-in Ethernet</string>
    </dict>
    <key>UserDefinedName</key>
    <string>Built-in Ethernet</string>
</dict>
```

Listing 3-2 shows what this dictionary looks like after the preferences monitor maps it.

Listing 3-2 Subset of service dictionary, after mapping into the dynamic store

```
/Network/Service/100
/Network/Service/100/IPv4
<dictionary>
    <key>ConfigMethod</key>
    <string>BootP</string>
</dictionary>

/Network/Service/100/Interface
<dictionary>
    <key>DeviceName</key>
    <string>en0</string>
    <key>Hardware</key>
    <string>Ethernet</string>
    <key>Type</key>
    <string>Ethernet</string>
    <key>UserDefinedName</key>
    <string>Built-in Ethernet</string>
</dictionary>

<dictionary>
    <key>UserDefinedName</key>
    <string>Built-in Ethernet</string>
</dictionary>
```

As you can see in Listing 3-2 (page 36), the IPv4 and Interface subdictionaries are now at the same level as their parent (the service dictionary) and have keys that describe their position in the parent dictionary. In addition, the empty DNS subdictionary in Listing 3-1 (page 36) is not represented in the mapping.

When it generates a new mapping, the preferences monitor keeps track of the differences from the old mapping. Applying these differences to the dynamic store triggers notifications associated with keys that have changed. This ensures that reconfiguration is automatic when, for example, a user selects a different location in Network preferences or from the Apple menu.

Kernel Event Monitor

The kernel event monitor maintains a list of all network interfaces defined in the system, the link status associated with each interface, and any assigned addresses. It monitors low-level kernel events and watches the network stacks, keeping track of the link status of each network interface. The kernel event monitor's main job is to post the status of each network interface in the dynamic store. This frees applications from having to reach into the kernel to find out, for example, if the Ethernet cable is plugged in or if the assigned addresses have changed.

Unlike most other agents, the kernel event monitor does not retrieve configuration information from `Setup:` keys in the dynamic store. Instead, it receives its input directly from the kernel and publishes its observations in some of the `State:` keys, as shown in Table 3-8.

Table 3-8 Keys the kernel event monitor publishes

Key	Property	Notes
State:/Network/Interface/*InterfaceName*/IPv4	Addresses	Published for all interfaces.
State:/Network/Interface/*InterfaceName*/IPv4	SubnetMasks	Published for broadcast interfaces.
State:/Network/Interface/*InterfaceName*/IPv4	BroadcastAddresses	Published for broadcast interfaces.
State:/Network/Interface/*InterfaceName*/IPv4	DestAddresses	Published for point-to-point interfaces.
State:/Network/Interface/*InterfaceName*/Link	Active	`TRUE` when interface is active.
State:/Network/Interface	Interfaces	This is an array of interface names.

IPv4 Configuration Agent

The IPv4 configuration agent configures Ethernet-type devices for IP networking. When the preferences monitor loads an updated mapping from the persistent store into the dynamic store, the IPv4 configuration agent receives a notification for each configured service. It gets the configuration information from the relevant `Setup:` keys and applies that configuration to the indicated interface. It then updates the dynamic store to reflect the actual IP addresses that were assigned. For some configurations, such as DHCP and BootP, the IPv4 configuration agent also updates the dynamic store with any configuration options it received from the server.

When a user switches locations or makes a change to the IP configuration preferences in Network preferences, the preferences monitor loads the new settings into the dynamic store. The IPv4 configuration agent receives notification of the changes and updates the network configuration according to the new settings. It then publishes the new network status information in the appropriate `State:` keys.

The IPv4 configuration agent also monitors changes in link status. It does this by registering interest in the `State:` keys the kernel event monitor publishes (shown in Table 3-8 (page 37)). This supports the automatic configuration of network interfaces when the computer is plugged into the network. Table 3-9 shows the keys the IPv4 configuration agent uses.

Table 3-9 Keys the IPv4 configuration agent uses

Key	Property	Notes
State:/Network/Interface	Interfaces	The list of network interfaces.
State:/Network/Interface/*InterfaceName*/Link	Active	The link status of a given interface.
Setup:/Network/Global/IPv4	ServiceOrder	An array of service IDs used for ranking and prioritization.
Setup:/Network/Service/*ServiceID*/Interface	DeviceName	The BSD interface name to be configured.
Setup:/Network/Service/*ServiceID*/Interface	Type	Configures services when Type is Ethernet.
Setup:/Network/Service/*ServiceID*/IPv4	ConfigMethod	Manual, BootP, DHCP, PPP, INFORM, LinkLocal.
Setup:/Network/Service/*ServiceID*/IPv4	Addresses	For Manual and INFORM, the IP address to be assigned.
Setup:/Network/Service/*ServiceID*/IPv4	SubnetMasks	For Manual and INFORM, the IP mask to be assigned.

The IPv4 configuration agent also publishes values in the dynamic store. Table 3-10 shows the keys the IPv4 configuration agent publishes and the circumstances under which it does so.

Table 3-10 Keys the IPv4 configuration agent publishes

Key	Property	Notes
State:/Network/Service/*ServiceID*/IPv4	Addresses	The IP address assigned for this service. For Manual and INFORM, this should be the same address specified in the `Setup:` keys.
State:/Network/Service/*ServiceID*/IPv4	SubnetMasks	The IP mask assigned for this service. For Manual and INFORM, this should be the same address specified in the `Setup:` keys.
State:/Network/Service/*ServiceID*/IPv4	Router	
State:/Network/Service/*ServiceID*/IPv4	InterfaceName	The BSD interface name associated with this service.

Key	Property	Notes
State:/Network/Service/*ServiceID*/DNS	DomainName	Published when ConfigMethod is DHCP, BootP, or INFORM and required data has been provided by the server.
State:/Network/Service/*ServiceID*/DNS	ServerAddresses	Published when ConfigMethod is DHCP, BootP, or INFORM and required data has been provided by the server.
State:/Network/Service/*ServiceID*/NetInfo	ServerAddresses	Published when ConfigMethod is DHCP, BootP, or INFORM and required data has been provided by the server.
State:/Network/Service/*ServiceID*/NetInfo	ServerTags	Published when ConfigMethod is DHCP, BootP, or INFORM and required data has been provided by the server.
State:/Network/Service/*ServiceID*/DHCP		DHCP-specific information, such as the lease time.

IPv6 Configuration Agent

Like the IPv4 configuration agent, the IPv6 configuration agent also configures Ethernet-type devices for IP networking. In addition, the IPv6 configuration agent configures FireWire devices and the new 6to4 interface. When the preferences monitor loads an updated mapping from the persistent store into the dynamic store, the IPv6 configuration agent receives a notification for each configured service. It gets the configuration information from the relevant `Setup:` keys and applies that configuration to the indicated interface. It then updates the dynamic store to reflect the actual IP addresses that were assigned.

The IPv6 configuration agent also monitors changes in link status. It does this by registering interest in the `State:` keys the kernel event monitor publishes (shown in Table 3-8 (page 37)). This supports the automatic configuration of network interfaces when the computer is plugged into the network. Table 3-11 shows the keys the IPv6 configuration agent uses.

Table 3-11 Keys the IPv6 configuration agent uses

Key	Property	Notes
State:/Network/Global/IPv4	PrimaryService	Used to determine which interface to use for 6to4 tunnelling.
State:/Network/Interface	Interfaces	The list of network interfaces.
State:/Network/Interface/*InterfaceName*/IPv6	Addresses	Manual or automatically assigned IP address.
State:/Network/Interface/*InterfaceName*/IPv6	Flags	Address-specifics flags.
State:/Network/Interface/*InterfaceName*/IPv6	PrefixLength	Length of address prefix or subnet.
State:/Network/Interface/*InterfaceName*/IPv6	Router	

Key	Property	Notes
State:/Network/Interface/*InterfaceName*/Link	Active	The link status of a given interface.
Setup:/Network/Service/*ServiceID*/IPv6	Addresses	The IP address to be assigned.
Setup:/Network/Service/*ServiceID*/IPv6	ConfigMethod	Automatic, Manual, RouterAdvertisement, 6to4.
Setup:/Network/Service/*ServiceID*/IPv6	Flags	Address-specific flags.
Setup:/Network/Service/*ServiceID*/IPv6	PrefixLength	Length of address prefix or subnet.
Setup:/Network/Service/*ServiceID*/IPv6	Router	Updates routing table with the default route.
Setup:/Network/Service/*ServiceID*/Interface	Type	Configures services when Type is Ethernet, FireWire or 6to4. Also configures IPv4 services over FireWire.
Setup:/Network/Service/*ServiceID*/6to4	Relay	6to4 relay address.

The IPv6 configuration agent also publishes values in the dynamic store. Table 3-12 (page 40) shows the keys the IPv6 configuration agent publishes and, for some keys, the circumstances under which it does so.

Table 3-12 Keys the IPv6 configuration agent publishes

Key	Property	Notes
State:/Network/Service/*ServiceID*/IPv6	Addresses	
State:/Network/Service/*ServiceID*/IPv6	Flags	
State:/Network/Service/*ServiceID*/IPv6	PrefixLength	
State:/Network/Service/*ServiceID*/IPv6	Router	When ConfigMethod is Manual.
State:/Network/Service/*ServiceID*/IPv6	InterfaceName	

IP Monitor

The main job of the IP monitor agent is to select the primary network service. This is typically the service that is associated with the default route and default DNS for the system. To make its determination, the IP monitor agent examines both the user's preferred priority of services and the current status of those services. It then selects the currently available service that is highest on the user's priority list and marks that service as primary in the dynamic store.

The IP monitor agent is driven by information other agents publish in the dynamic store. It monitors the user's preferences mapped in by the preferences monitor and the configuration status of services published by the IPv4 and IPv6 configuration agents and the PPP controller. Table 3-13 shows the keys the IP monitor agent uses.

Table 3-13 Keys the IP monitor uses

Key	Property	Notes
State:/Network/Interface/*InterfaceName*/Link	Active	
Setup:/Network/Global/IPv4	ServiceOrder	
Setup:/Network/Service/*ServiceID*/IPv4	Router	Specified by the user. In general, these settings are preferred over values derived from the network.
Setup:/Network/Service/*ServiceID*/DNS	DomainName	
Setup:/Network/Service/*ServiceID*/DNS	ServerAddresses	
Setup:/Network/Service/*ServiceID*/DNS	SearchDomains	
Setup:/Network/Service/*ServiceID*/DNS	SortList	
Setup:/Network/Service/*ServiceID*/NetInfo	BindingMethods	
Setup:/Network/Service/*ServiceID*/NetInfo	ServerAddresses	
Setup:/Network/Service/*ServiceID*/NetInfo	ServerTags	
Setup:/Network/Service/*ServiceID*/Proxies	All properties	
State:/Network/Service/*ServiceID*/IPv4	Addresses	Settings derived from the type of configuration. In general, these settings are used when a user-specified setting is unavailable.
State:/Network/Service/*ServiceID*/IPv4	Router	
State:/Network/Service/*ServiceID*/DNS	DomainName	
State:/Network/Service/*ServiceID*/DNS	ServerAddresses	
State:/Network/Service/*ServiceID*/DNS	SearchDomains	
State:/Network/Service/*ServiceID*/DNS	SortList	
State:/Network/Service/*ServiceID*/NetInfo	ServerAddresses	
State:/Network/Service/*ServiceID*/NetInfo	ServerTags	
State:/Network/Service/*ServiceID*/Proxies	All properties	

The IP monitor also publishes values in the dynamic store. Table 3-14 shows the keys the IP monitor publishes and the circumstances under which it does so.

Table 3-14 Keys the IP monitor publishes

Key	Property	Notes
State:/Network/Global/IPv4	PrimaryService	Identifies which network service is deemed primary.
State:/Network/Global/IPv4	PrimaryInterface	Identifies which network interface is deemed primary.
State:/Network/Global/IPv4	Router	Updates routing table with the PrimaryService default route. Handles "proxy arp" which is enabled if the Router value is the same as one of the service's Addresses values (enabled automatically if Router has no value).
State:/Network/Global/DNS	DomainName	Checks each of the DNS properties for the primary service, favoring a property from the `Setup:` key over one from the `State:` key.
State:/Network/Global/DNS	ServerAddresses	
State:/Network/Global/DNS	SearchDomains	
State:/Network/Global/DNS	SortList	
State:/Network/Global/NetInfo	ServerAddresses	Generates the list of NetInfo server addresses and server tags based on the primary service's NetInfo information.
State:/Network/Global/NetInfo	ServerTags	
State:/Network/Global/Proxies	All properties	If available, uses the primary service's proxy information in the `Setup:` key, otherwise, uses the proxy information in the `State:` key.
State:/Network/Service/*ServiceID*/IPv6	Router	For ConfigMethod other than Manual.

PPP Controller

The PPP controller configures PPP interfaces for IP networking. It manages connections through dial-up modems and through PPP over Ethernet (PPPoE) and VPN (virtual private network) connections using PPTP (point-to-point tunneling protocol) and L2TP (layer 2 tunneling protocol). The PPP controller interacts with dialer applications and instantiates PPP interfaces as needed.

The PPP controller gets its configuration information from the `Setup:` keys provided by the preferences monitor and receives notification when these keys change. After an interface is configured, the PPP controller publishes the IP, destination, and router addresses and the DNS information provided by the PPP server. Table 3-15 shows the keys the PPP controller uses.

Table 3-15 Keys the PPP controller uses

Key	Property	Notes
Setup:/Network/Service/*ServiceID*/Interface	DeviceName	
Setup:/Network/Service/*ServiceID*/Interface	Type	Configures services with Type PPP
Setup:/Network/Service/*ServiceID*/Interface	SubType	PPPSerial, PPPoE, PPTP, L2TP
Setup:/Network/Global/IPv4	ServiceOrder	
Setup:/Network/Service/*ServiceID*/PPP	All properties	

The PPP controller publishes many keys in the dynamic store. Table 3-16 shows some of them.

Table 3-16 Some keys the PPP controller publishes

Key	Property	Notes
State:/Network/Service/*ServiceID*/IPv4	Addresses	
State:/Network/Service/*ServiceID*/IPv4	DestAddresses	
State:/Network/Service/*ServiceID*/IPv4	Router	
State:/Network/Service/*ServiceID*/IPv4	InterfaceName	
State:/Network/Service/*ServiceID*/DNS	DomainName	
State:/Network/Service/*ServiceID*/DNS	ServerAddresses	
State:/Network/Service/*ServiceID*/DNS	SearchDomains	
State:/Network/Service/*ServiceID*/DNS	SortList	
State:/Network/Service/*ServiceID*/PPP		Reserved for Apple use

Using the Schema

The previous section describes how the System Configuration schema structures the persistent store and which key-value combinations define sets and specific services. This is essential knowledge for any developer writing an application that defines a set or provides a network service, because the information must be presented in the correct format. It is also essential knowledge for the developer of an application that requires detailed PPP connection information or that requests notifications. This is because the configuration agents and the dynamic store also adhere to the schema. Whether you need to register for notifications or get information about current connection status, you need to know which key-value pairs hold the data you're interested in.

This section describes three situations in which knowledge of the schema is essential.

Programmatically Setting Preferences

The System Configuration framework provides an API to programmatically set the keys and values that define sets and services. Unfortunately, the API is very low level. For the most part, it consists of functions to get and set individual key-value pairs. It provides no guidance on how to lay out a set or network service dictionary and no support for building these complex structures.

To help meet the needs of developers, Apple's Developer Technical Support provides a large library of sample code that wraps some of the low-level System Configuration API in higher-level functions. Not only does this library insulate you from much of the low-level manipulation of key-value pairs, it also provides a streamlined template you can use to define a basic service. When you get this basic service working, you can then tweak the settings, using the information in"Layout of the Persistent Store" (page 23) as a guide.

The sample code library, called *MoreSCF*, contains several modules that help you programmatically define sets, find active ports, and activate services.

Getting Detailed PPP Connection-Status Information

The System Configuration framework provides API that allows you to get detailed information about the current PPP connection. Available in Mac OS X version 10.3 and later, this API gives you access to information gathered by the PPP controller. To use this API, you should be familiar with the layout of the network service dictionary and PPP protocol entity dictionary, as defined by the schema. This is because the information the API returns to you follows the same structure.

Recall that the preferences monitor reads the persistent store and populates the dynamic store with flattened keys and values that describe the user's currently active configuration. This information is in the setup portion of the dynamic store, using keys prefixed by `Setup:` (for more information on how these keys are created, see "Preferences Monitor " (page 35)). The other configuration agents read the setup keys they're interested in, monitor sources of network status information, and publish the results of their observations and configuration actions in the state portion of the dynamic store. Although it is tempting to go directly to the dynamic store and get the value of a specific key, in some cases, it's not as effective as you might expect.

In particular, to get current PPP status, it's much better to use the `SCNetworkConnectionCopyExtendedStatus` function in the SCNetworkConnection API (defined in the `SCNetworkConnection.h` file in the System Configuration framework). Using this function, you can get very detailed connection-status information from the PPP controller, including some information that is never copied into the state portion of the dynamic store.

For example, when a user chooses to make a PPP connection through a modem, there are several steps the modem takes before connection is finally established. If you register for notification on the PPP Status key in the dynamic store (`State:/Network/Service/serviceID/PPP/Status`), you'll receive notification when this key's value is updated. If you need connection-status information sooner, you might choose to retrieve the value of the PPP Status key immediately after a connection is initiated. When you do this, however, you'll receive the value Disconnected until after the connection completes. This is because the PPP controller does not update the dynamic store PPP Status key until after the connection is complete. If, instead, you use the `SCNetworkConnectionCopyExtendedStatus` function to request immediate notification, you'll be able to observe intermediate states, such as Initialize, Connect, Negotiate, Authenticate, and Connected. In this way, you can get much more detailed (and more accurate) information than if you simply watched the dynamic store.

It's important to remember that the SCNetworkConnection API, like most other System Configuration APIs, depends on the schema-defined layout of network-service dictionaries and of the dynamic store. When you use the `SCNetworkConnectionCopyExtendedStatus` function to get PPP connection-status information, you receive a dictionary that contains a subdictionary for each of the service's subcomponents, such as PPP, IPv4, and Modem. To successfully interpret the information in these dictionaries, you need to know how the schema defines the layout of the key-value pairs. For more information on the structure of these dictionaries, see the appropriate sections in "The NetworkServices Dictionary" (page 26).

Getting Notifications

Understanding how the schema positions the information in the dynamic store gives you a lot of power. In particular, it gives you access to a great deal of low-level network status information. On the other hand, it also fosters the notion that you should be paying attention to every small change in network status. Conceivably, you could write an application that watches large numbers of keys and provides the user with constant updates on every network status change. This is seldom necessary, however, and it is not advisable.

Although the System Configuration framework enables you to request information at this level, it also provides APIs that abstract some of it and provide it in more palatable forms. Therefore, it's important to know when it's appropriate to request notifications on individual keys and when it's appropriate to use the API to get information.

For example, if your application needs to be aware of changes to network proxy settings while it runs, you should watch the dynamic store key associated with the network proxy settings. This is will ensure you receive a notification when any of these settings change. If, instead, you call the `SCDynamicStoreCopyProxies` function to get the proxy settings in force when your application starts, you won't get notified of changes as your application runs.

On the other hand, watching configuration keys (the `Setup:` keys in the dynamic store) is usually a bad idea unless you're developing a configuration agent. This is because your application has no way of knowing if, or more importantly when, configuration changes are reflected in the system's running configuration.

Determining Reachability and Getting Connected

This chapter describes the System Configuration reachability and connection APIs. You use the reachability API to determine if data destined for a remote host can leave the local machine. To start or stop a PPP connection, you use the connection API.

Note: The System Configuration connection API supports only PPP-based connections; it does not not help you establish a TCP-based connection.

It is not necessary to have an in-depth understanding of the System Configuration architecture to determine reachability or start a PPP connection. The reachability and connection APIs provide a layer of abstraction that allows an application to make connections without worrying about the state of the network stack.

Scope of the Reachability and Connection APIs

Many types of applications need to find out if a remote host is reachable or to start a PPP-based connection. For example, an application might need to check for software updates that are available from a remote server. In Mac OS X version 10.3 and later, the System Configuration framework provides APIs to accomplish these tasks quickly and easily, without requiring a detailed knowledge of System Configuration architecture. In particular, the reachability and connection APIs do much of their work behind the scenes, so the application does not have to poll for status changes or watch specific keys in the dynamic store.

The System Configuration reachability API helps an application determine if a remote host is reachable. A remote host is considered reachable if a data packet sent to the host can leave the local computer, regardless of what ultimately happens to the packet. In other words, the reachability of a remote host does not guarantee that the host will receive the data.

In practice, when a remote host is deemed reachable, but the packets you send to it fail to arrive, the myriad possible reasons for the failure fall into two broad categories:

1. A part of the Internet connection over which you have no control is broken. For example, the remote host's server is down.
2. A part of your local network infrastructure over which you might have control is broken. For example, your modem hasn't dialed or your AirPort base station is turned off.

The reachability API cannot help you with problems in the first category. As long as data packets can leave the local machine, the remote host is considered reachable. What the reachability API can provide is help diagnosing some of the problems in the second category. If, for example, the modem is currently disconnected, the API can tell you this.

To further define the scope of the System Configuration reachability and connection APIs, this document uses the following phrases to distinguish between two different types of network connection:

- **Network transport connection**. This is a TCP-based connection you initiate using, for example, BSD sockets.
- **Network link connection**. This is a PPP-based connection you initiate by, for example, telling the modem to dial.

The reachability API helps you determine if a remote host is reachable by examining the status of the local network link connection. The connection API allows you to start a network link connection. After you successfully start a network link connection, you use different API (such as Core Foundation networking API) to establish a network transport connection.

A Reachability and Connection Example

Determining reachability and requesting a network link connection often go hand in hand. As an example, consider the running of an email application. The sequence of events might go like this:

1. The user launches the email application to check her email.
2. The email application uses the reachability API to find out if the POP server is reachable.
3. The reachability API tells the application that the POP server is reachable, but that a network link connection must first be made.
4. The email application asks the user if she wants to dial the modem and she clicks "OK". (The application might skip this step if the user has set a preference that tells the modem to dial automatically when the email application is launched.)
5. The email application uses the connection API to start a PPP connection, which causes the modem to dial. At this point, the email application can return to its work.
6. The reachability API notifies the email application that the network link connection is up and the email application then uses a network transport connection to fetch the email.

After the email application delivers the email to the user, it might or might not cause the PPP connection to drop, depending on the user's preferences. Either way, when the email application quits, the PPP connection is dropped and, if the email application held the last reference to the connection, the modem disconnects. For more information on connection references, see "Starting and Stopping a Connection" (page 53).

Using the Reachability API

Note: Function names in the System Configuration framework APIs follow Core Foundation function-name conventions. In particular, a function that has "Create" or "Copy" in its name returns a reference you must release. To learn more about Core Foundation memory management policies, see *Memory Management Programming Guide for Core Foundation*.

In versions of Mac OS X prior to 10.3, the System Configuration reachability API consisted of two functions. The functions, `SCNetworkCheckReachabilityByAddress` and `SCNetworkCheckReachabilityByName`, supply information about the reachability of a remote host by providing a set of flags. The flags (defined in `SCNetwork.h`) indicate, for example, that the specified remote host could be reached using the current network configuration. Unfortunately, these functions encouraged polling because they did not support notifications.

In Mac OS X version 10.3, the System Configuration framework introduced the SCNetworkReachability API which supports asynchronous notifications of changes in the connection to a remote host. You still get the same set of connection-status flags the previous reachability functions supplied to determine if a connection is required, but you no longer have to poll to find out if a connection attempt is successful.

Most functions in the new API rely on an `SCNetworkReachabilityRef` (defined in `SCNetworkReachability.h`) you use to identify the remote target. You create this reference for a specific address or hostname and use it to add the reference to your run loop and set up a callback function. You can use the information in the connection-status flags to request a network link connection and the new reachability API notifies you when the connection is established.

You can use the new reachability functions in your application to perform the following tasks:

- Create a reference for your target remote host you can use in other reachability functions.
- Add the target to your run loop.
- Provide a callback function that's called when the reachability status of your target changes.
- Determine if the target is reachable.

The following sections describe how to use the SCNetworkReachability functions to perform these tasks. For a code example demonstrating how to use the SCNetworkReachability API, see *SimpleReach*.

Creating a Reference

The SCNetworkReachability API provides three functions that create a target reference you can use to monitor the reachability of a remote host:

- `SCNetworkReachabilityCreateWithAddress`
- `SCNetworkReachabilityCreateWithAddressPair`
- `SCNetworkReachabilityCreateWithName`

If you are interested in the reachability of only the remote host, you can use either `SCNetworkReachabilityCreateWithAddress` or `SCNetworkReachabilityCreateWithName`. For `SCNetworkReachabilityCreateWithAddress`, you supply a remote host address in a `sockaddr` structure. For more information on the `sockaddr` structure, see the networking `man` page at http://developer.apple.com/documentation/Darwin/Reference/ManPages/man4/netintro.4.html. For `SCNetworkReachabilityCreateWithName`, you supply a remote host name, such as www.apple.com.

If you need to monitor possible changes to both the local and remote host addresses, you use the `SCNetworkReachabilityCreateWithAddressPair` function. This function returns a reference you can use to find out if:

- The address of your local host changes
- The remote address becomes unreachable
- The network route associated with the remote host changes

All three functions return an object of type `SCNetworkReachabilityRef` for the remote host, which you can use in any of the other SCNetworkReachability functions.

Adding a Target to a Run Loop

To ensure that your application receives notification when the reachability status of the target remote host changes, you add the target's `SCNetworkReachabilityRef` to your application's run loop. A run loop monitors sources of input to an application. When an input source becomes ready for processing (because some activity occurs), the run loop dispatches control to a callback function associated with the input source. For more information on run loops and modes, see *Run Loops*.

For a network-aware application, the input source is the `SCNetworkReachabilityRef`, the activity is a change in the target's connection status, and the callback function is one you provide. Thus, by using the reachability API to add your target to your application's run loop and to provide a callback function (described in "Associating a Callback Function With the Target" (page 50)), you ensure your application will be notified of changes in the target's reachability.

To add an `SCNetworkReachabilityRef` to your application's run loop, you use the `SCNetworkReachabilityScheduleWithRunLoop` function. You provide the `SCNetworkReachabilityRef`, a reference to your application's run loop, and the mode in which the run loop should run, typically the default mode.

To remove an `SCNetworkReachabilityRef` from your application's run loop, you use the `SCNetworkReachabilityUnscheduleFromRunLoop` function, which requires the same parameters as the `SCNetworkReachabilityScheduleWithRunLoop` function.

Associating a Callback Function With the Target

To make use of the notifications the reachability API provides, you should associate a callback function with the `SCNetworkReachabilityRef` representing the remote host you're interested in. The callback function might display the change in the connection to the user or perform some other task.

To associate a callback function with your target, you first define a function of type `SCNetworkReachabilityCallBack` (this type is defined in `SCNetworkReachability.h`). To pass contextual information about the changes to your callback function, you define a structure of type

SCNetworkReachabilityContext. You use the info argument to contain the context information (you can also specify NULL, if context information is unnecessary). Finally, you pass the callback function, the SCNetworkReachabilityRef representing the remote host, and the context to the SCNetworkReachabilitySetCallback function.

Note: If you pass NULL for the SCNetworkReachabilityCallBack parameter, you remove the callback function currently associated with the target.

Determining Reachability

The SCNetworkReachability API provides the SCNetworkReachabilityGetFlags function you can use to determine the reachability of a remote host. This function supplies the same connection-status flags the older reachability functions defined in SCNetwork.h supplied.

To use this function, you supply the SCNetworkReachabilityRef for your remote host and the address of the variable you declare to contain the flags. The flags you can receive are listed in Table 4-1.

Table 4-1 SCNetworkConnectionFlags

Flag name	Meaning
kSCNetworkFlags-TransientConnection	The target is reachable through a transient connection (for example, PPP).
kSCNetworkFlagsReachable	The target is reachable using the current network configuration.
kSCNetworkFlags-ConnectionRequired	The target is reachable using the current network configuration, but a connection must be established first. For example, a suitable dialup connection exists, but it is not active.
kSCNetworkFlags-ConnectionAutomatic	The target is reachable using the current network configuration, but a connection must be established first. Further, any traffic directed to the target will automatically initiate the connection.
kSCNetworkFlags-InterventionRequired	The target is reachable using the current network configuration, but a connection must be established first. Further, some user action is required to establish connection (for example, providing a password).
kSCNetworkFlags-IsLocalAddress	The target is associated with a network interface on the current system. For example, the target address is one of the IP addresses assigned to the system.
kSCNetworkFlagsIsDirect	Network traffic to the target will not pass through a router because the destination address is on a network that's directly connected to one of the local machine's interfaces (for example, it's on the same subnet as the local machine).

Using the Network Connection API

In Mac OS X version 10.3, the System Configuration framework introduced the SCNetworkConnection API. This API allows an application to control connection-oriented services already defined in the system. Currently, an application can control only PPP services with this API.

In addition to controlling an existing service, an application can use the SCNetworkConnection API to get information about the connection. The API provides connection-status information on two levels:

- High-level, generic information that describes the status of the network connection, such as connected or disconnected
- Detailed, PPP-specific information that describes the status of the PPP stack

These two levels of status information target different types of applications. A network-aware application might want to display whether or not a network link connection is live, but is probably uninterested in knowing if the PPP controller is currently configuring the link layer. A complex dialer application, on the other hand, might need to know exactly what the PPP controller is doing at each step of the connection process.

Although these functions are defined in the same header file, this chapter does not describe the PPP-specific connection-status functions in `SCNetworkConnection.h`. For more information on how to use these functions and how to use the System Configuration schema to interpret the results, see "The System Configuration Schema" (page 23). Instead, this chapter focuses on the remainder of the SCNetworkConnection API, describing how an application can use it to accomplish the following tasks:

- Create a reference representing the connection you can use with other connection functions.
- Add the connection reference to your application's run loop.
- Start a connection.
- Get the status of a connection.
- Stop an existing connection.

For a code example demonstrating the use of the SCNetworkConnection API, see *SimpleDial*.

Creating a Connection Reference

The SCNetworkConnection API provides one function you can use to create a connection reference: `SCNetworkConnectionCreateWithServiceID`. To use this function, you supply a service ID and a callback function, along with a couple of other parameters, and you receive an object of type `SCNetworkConnectionRef`. You use this object to represent the connection in the other SCNetworkConnection functions.

Because you'll be using the `SCNetworkConnectionRef` to refer to a specific PPP connection, you must supply a unique service ID to identify it. There are two ways to get this service ID. One way is to look in the dynamic store for available services and choose one. The second way is for your application to use the `SCNetworkConnectionCopyUserPreferences` function to get the default service ID (the one the Internet Connect application uses).

Although you can pass NULL for the callback function parameter, it is not recommended. If you don't define a callback function, your application will not receive status-change notifications and will have to poll for updates.

Adding a Connection Reference to a Run Loop

To ensure your application gets notified of changes in a specific connection's status, you can add the `SCNetworkConnectionRef` representing the connection to your application's run loop. When the connection's status changes, the `SCNetworkConnectionRef` alerts the run loop and the run loop passes control to the callback function you provide.

To add an `SCNetworkConnectionRef` to your application's run loop, you use the `SCNetworkConnectionScheduleWithRunLoop` function. You provide the `SCNetworkConnectionRef`, a reference to your application's run loop, and the mode in which the run loop should run (in most cases, the default mode). For more information on run loops and modes, see Introduction to Run Loops in the Core Foundation Reference Library.

To remove the `SCNetworkConnectionRef` from your application's run loop, you use the `SCNetworkConnectionUnscheduleFromRunLoop` function. Like the `SCNetworkConnectionScheduleWithRunLoop` function, this function expects the `SCNetworkConnectionRef`, a reference to your application's run loop, and the run loop mode.

Starting and Stopping a Connection

The SCNetworkConnection API provides two functions your application can use to control a connection using an `SCNetworkConnectionRef` object:

- `SCNetworkConnectionStart`
- `SCNetworkConnectionStop`

Both functions return immediately while the connection or disconnection process they initiate proceeds asynchronously. If you add the connection's `SCNetworkConnectionRef` to your application's run loop and provide a callback function, your application is notified when the status of the connection changes. Your application can then check the status to determine if the connection process is complete. If you don't add the connection reference to the run loop, you will have to poll to discover the connection status, and this is not recommended.

By default, the `SCNetworkConnectionStart` function uses the user's preferred connection settings to start the connection. You can, however, provide a dictionary of values to override some of these settings for the duration of the connection.

If you do provide a dictionary of additional settings be aware that:

- The dictionary must be in the correct format for a network service dictionary (described in "The NetworkServices Dictionary" (page 26)). If you do not follow this format precisely, the PPP controller may ignore the dictionary.
- The PPP controller merges the settings you provide with the user's existing settings before the connection is established, ignoring any inappropriate values in your dictionary.

Starting and stopping a connection are implicitly arbitrated. This means that the interest other applications may have in a connection is taken into account before the connection is stopped. For example, application "B" can start a connection that has already been started by application "A". Application "A" may choose to stop the connection but the system should not stop the connection until "B" is finished with it. Using the SCNetworkConnection functions, application "A" can call `SCNetworkConnectionStop` on the connection and the function will return success, but the connection will not stop until "B" calls `SCNetworkConnectionStop`. An application can also use the `linger` parameter (discussed below) to register interest.

The `SCNetworkConnectionStart` function allows an application to indicate an interest in a connection. In this way, an application can request that the existence of the connection be tied to actions the application takes. In the simplest case, an application might want a connection to start when the application calls `SCNetworkConnectionStart` and stop when any of the following events occur:

- The application quits.
- The application releases the `SCNetworkConnectionRef` representing the connection.
- The application calls `SCNetworkConnectionStop`.

To indicate interest in a connection, an application can use the `linger` parameter of the `SCNetworkConnectionStart` function. An application, such as an email client, that needs to get connected, perform its tasks, and get disconnected, sets the parameter to `FALSE`. This indicates that the connection should stop when the application quits or releases the connection reference. A PPP dialer application, on the other hand, might choose to set the parameter to `TRUE` so that the user has control over the modem. The `TRUE` value indicates the connection should *not* stop when the application quits or releases the connection reference.

It's important to note, however, that several concurrent applications might register interest in the same connection. When this is the case, the System Configuration framework keeps track of the references to the connection, stopping it when the last reference is released (or the last application holding a reference quits).

The `SCNetworkConnectionStop` function performs an arbitrated stop of the connection. In other words, it closes the connection if:

- There are no other interested applications currently running or holding references to the connection. If there are, the `SCNetworkConnectionStop` function returns success to the calling application, but the connection will persist until all interested applications terminate or release their references to the connection.
- An application calls `SCNetworkConnectionStop` and passes `TRUE` in the `forceDisconnect` parameter. This stops the connection regardless of the interest of other currently running applications. Most applications do not need to force a connection to stop in this way. A PPP dialer application, however, would probably choose to do this because it ensures that the connection stops when the user wants it to.

Getting the Status of a Connection

An application can use the `SCNetworkConnectionGetStatus` function to get the high-level status of the connection. You pass the `SCNetworkConnectionRef` representing the connection to the function and you receive one of the constants shown in "Introduction to System Configuration Programming Guidelines" (page 7).

Table 4-2 High-level connection status values

Status	Meaning
`kSCNetworkConnectionInvalid`	The network connection refers to an invalid service.
`kSCNetworkConnectionDisconnected`	The network connection is disconnected.
`kSCNetworkConnectionConnecting`	The network connection is connecting.
`kSCNetworkConnectionConnected`	The network connection is connected.
`kSCNetworkConnectionDisconnecting`	The network connection is disconnecting.

If your application needs to know more about the PPP connection, such as when the PPP controller is authenticating to the server, you should use the `SCNetworkConnectionCopyExtendedStatus` function. You pass the `SCNetworkConnectionRef` representing the connection to the function and you receive a dictionary that contains subdictionaries for each subcomponent of the service, such as PPP, IPv4, and Modem. The PPP connection status is represented by a constant defined in the SCNetworkConnection API. Table 4-3 (page 55) shows the current set of constants. Note that additional status values might be defined in the future, so your application should be able to handle an unknown value.

Table 4-3 PPP connection status values

Status	Meaning
`kSCNetworkConnectionPPPDisconnected`	PPP is disconnected.
`kSCNetworkConnectionPPPInitializing`	PPP is initializing.
`kSCNetworkConnection-PPPConnectingLink`	PPP is connecting the lower layer (as when, for example, the modem is dialing out).
`kSCNetworkConnection-PPPDialOnTraffic`	PPP is waiting for networking traffic to automatically establish the connection.
`kSCNetworkConnection-PPPNegotiatingLink`	The PPP lower layer is connected and PPP is negotiating the link layer (the LCP protocol).
`kSCNetworkConnection-PPPAuthenticating`	PPP is authenticating to the server (using PAP, CHAP, MSCHAP, or EAP protocol).
`kSCNetworkConnection-PPPWaitingForCallBack`	PPP is waiting for the server to call back. *Note*: this state is defined but will not occur because CallBack is not supported in Mac OS X version 10.3.
`kSCNetworkConnectionPPPNetgotiating-Network`	PPP is authenticated and is now negotiating the networking layer (using IPCP or IPv6CP protocol).
`kSCNetworkConnectionPPPConnected`	PPP is fully connected for at least one networking layer. Additional networking protocols might still be negotiating.
kSCNetworkConnectionPPPTerminating	PPP networking and link protocols are terminating.

Status	Meaning
`kSCNetworkConnection-PPPDisconnectingLink`	PPP is disconnecting the lower connection layer (as when, for example, the modem is hanging up).
`kSCNetworkConnection-PPPHoldingLinkOff`	PPP is disconnected and maintaining the link temporarily "off".
`kSCNetworkConnectionPPPSuspended`	PPP is suspended as the result of the "suspend" command as when, for example, a V92 Modem is "On Hold".
`kSCNetworkConnection-PPPWaitingForRedial`	PPP found a busy server and is waiting for redial.

For more information on why you might choose to use the `SCNetworkConnectionCopyExtendedStatus` function, see "Getting Detailed PPP Connection-Status Information" (page 44).

Document Revision History

This table describes the changes to *System Configuration Programming Guidelines*.

Date	Notes
2006-02-07	Removed link to unavailable, legacy AppleTalk document; added links to sample code.
2004-11-02	Added link to API reference documentation.
2004-04-22	First version.

Document Revision History