Sessions and Personalization

We know that HTTP is stateless. In other words, each HTTP request is completely independent of every other request, at least as far as HTTP is concerned. Yet, it is frequently crucial to carry state information between HTTP requests.

Example 1 Shopping Session
Consider a shopping cart. If you put a DVD in your shopping cart in one HTTP request, you expect to find it still in your shopping cart when you make your next HTTP request two minutes later. See Fig. 1. Somehow, the web site has to make the connection between these two requests, even though it may have serviced hundreds of requests from other users in the mean time.

If the state is not maintained in the transfer protocol, then the state has to be maintained at "the application level", above the transfer protocol. In other words, we need some mechanism, above HTTP, that remembers appropriate information from previous requests to be able to interpret the current request in context. In this chapter, we will introduce the concept of *sessions* used for this purpose, describe *cookies* used to implement sessions, and discuss the possibility of personalization for web interaction.

TCP Sessions: Application-level sessions, which are the subject of this chapter, are quite different from TCP sessions, which we discuss briefly in this sidebar.
After processing each HTTP request, the server and the client *may* close the TCP connection. Since even a single URL fetch can result in multiple HTTP requests (due to embedded graphics and the like), there will frequently be a whole sequence of HTTP requests between the same client-server pair, and in this case it is more efficient just to leave the connection open.

Multiple HTTP requests can thus be serviced within a single TCP "session", amortizing the overhead of TCP set up over these. If it is desired that this connection not be closed, either side could add a
Connection: Keep-Alive
to the HTTP header. Conversely, to "force" closure of connection, either side could add a
Connection: Close
Of course, the other party is not obligated to follow these requests. For instance, it may make sense for an overloaded server to ignore "keep-alive" requests from clients. A browser often does not know whether additional HTTP requests will be made to the same site, since that largely depends on what the user clicks on next. So the browser does not have a strong basis on which to make a good prediction regarding the desirability of keeping the TCP connection open. Usually, keeping an unnecessary connection open is of little cost to the client, but a whole lot of idle connections can collectively impose a significant load on the server. In consequence, many browsers will routinely request that the connection be kept alive, and many servers will routinely ignore such requests, instead implementing a time out policy to close TCP connections. We see an instance here of a mechanism that permits cooperating parties to play "nice," while selfish behavior and lack of trust result in TCP session policies that are less than ideal.
In short, TCP "sessions" represent an attempt at performance optimization, and are not the application-level sessions of interest in this chapter.

1 Sessions

When we say sessions must be maintained by an "application" above the HTTP level, the immediate question is which application? There is nothing standard that runs on top of HTTP. There is no universal protocol for this purpose. Instead, responsibility rests with the web server to maintain session state.

State is normally saved through the use of *session variables.* Values can be set for these variables during one request, and then read during a subsequent request in the same session. If all requests for a session came in sequence, with no intervening other requests, maintaining state with session variables would be straightforward. Things get more complicated since we require session maintenance across multiplexed requests from many different clients. Let us look at PHP session facilities next, as an instance of how sessions could be managed on a web server, and how multiple sessions can be maintained open in parallel. Later we will consider how to know which session a particular HTTP request belongs to.

Whenever PHP is invoked in the context of a session (we will discuss below how such context is established), the PHP script has access to a set of session variables. Think of the concept of session as similar to that of a file as far as programming is concerned. You have to start a session explicitly (like an open file), and close a session explicitly when done. Similarly, you have to associate each session variable with the session, allocating storage for this variable in the session structure, and must delete each session variable when it is no longer required. See Fig. 2.

Many server programming languages, such as PHP, take reasonable default actions if you do not initialize and manage your variables properly: new sessions will be started without you explicitly asking for one, new variables will be defined upon first use, all variables will be deleted at session end, and so forth. The default value for a variable in PHP not yet assigned a value (whether session variable or regular variable) is an empty string. If you rely on these facilities, you may not observe some of these actions that happen "under the covers".

Having session variables defined in a semi-permanent session structure is easy – we have an instance of the session structure for each active session, and we identify it by means of a session ID. The remaining challenge is to establish this session context, and the only mechanism we have is to use HTTP. There are two aspects to this, we must decide when to begin and end a session, and we must identify the session to which each request belongs. In other words, at the time of PHP script invocation at the web server, we must specify whether there is a session context and, if so, which one. Unfortunately, we cannot rely upon getting something as convenient as explicit commands to "begin session" and "end session". Instead, we have to decide when to take these actions. Typically, a new session is begun whenever a request is not associated with an existing session. Session end is more problematic. An explicit "exit" or "logout" request would certainly do it, if these are available. A time out is the usual method.

When setting a time out, would you count time from the beginning of session (first request) or from the most recent request? The former imposes a limit on the total length of session, the latter on the period of inactivity between requests in a session. In most, but not all, applications, the latter is likely to be a more meaningful choice.

For each HTTP request, we have to identify which of possibly multiple sessions currently maintained on the server to associate it with, if any. One possible place we could include this information is the URL, and we discuss this next. A second possibility is to use a special header field called *cookie*, which we will describe later in a subsequent section.

Recall that the URL allows an optional argument part at the end. The interpretation of this argument part is left completely up to the server. Therefore, it is straightforward to enhance the URL of the request with the session id encoded in this argument part. So we are set as long as the client will submit requests with the proper URL extension. One reasonable way to accomplish this is as follows (See Fig. 3):

The first request from a user is made to the regular (unenhanced) URL, and begins a session. The page served in response has a session encoding extension applied to all embedded URLs in it, other than any pointing to external sites. When any of these links is followed, the client request now includes the session encoding extension in the URL. The server is able to associate the request with the correct session. The page served in response again has the session encoding extension applied to all embedded internal URLs. Thus, as long as successive pages are reached by clicking on links, each requested URL will have the session identifier encoded in it. If the user actually types the URL in, or obtains it afresh from some referral place, such as a search results page, then the session identifier will be missing, and the user will be placed in a new session.

In addition to URLs that are actually present on a web page in the form of links, there could be dynamically generated URLs, for instance as the result of filling in and submitting a form. The same session-identifying extension must be applied to these URLs as well. (Filled in forms are usually submitted using the HTTP POST method, allowing a content part, in which considerable identifying information could be filled in. This space could be used in preference to a URL extension. Unless we have reason to expect all requests from a client to use the POST method, there is no point switching from the URL-encoding that we need for requests using other methods.)

URLs may be bookmarked, and may be used at a later time. If the encoded session identifier is no longer valid, then the request is treated as if it came with no session identifier, and a new session is established. Note that this means session identifiers must not be recycled too soon. We want to make it extremely unlikely that a stale session identifier will inadvertently match the identifier of a current ongoing session with some unrelated different user. See Fig. 4.

Furthermore, session identifiers should not simply be generated in sequence, or in any other pattern that is easy to observe and replicate. Otherwise, an attacker may manufacture a fake session identifier that is likely to match a current open session. Through this means, the attackers requests get intermingled with the real user's requests as part of the same session. At the very least, this can confuse the real customer. Worse outcomes are possible, if there are other security holes.

2 Log In

Notice that the establishment of a session requires no personal identifying information on the part of the user. The benefits of state maintenance during the session can be made available to all visitors to the website, while permitting them to remain anonymous.

While anonymous sessions suffice for many purposes, for certain actions, such as the sale of an item, we may wish to authenticate the user.

There is considerable work on anonymous transactions. It is possible, using one of several alternative schemes devised for this purpose, to make a purchase on the web, and pay for it, yet remain completely anonymous. We cannot describe the full details of these schemes here. See [anonymous e-cash]. There are two major issues. The first is the delivery of HTTP requests, and routing back of responses, without revealing the location or identity (IP address) of the requester. The second is the creation of an electronic equivalent of cash that permits payments to be made with anonymity. These anonymity techniques are limited to the virtual world, and so anonymity does not carry over when physical goods have to be shipped.

The standard technique to authenticate a user is by means of a user name and password. The web server can maintain state on a per user basis, in addition to or instead of, on a per session basis. Once we have information identifying the user, the effect of "session variables" can be obtained through state stored in the user "record". A significant advantage is that there is no need to time out sessions. For example, it is quite alright for a user to place some items in a shopping cart and go away for several days. When shopping is resumed, the user should find the shopping cart as it was left, with the items placed in it days earlier.

HTTP requests from a user who is logged in must be distinguished from anonymous requests. The issues with regard to communicating (authenticated) user identification are very similar to the issues with regard to communicating session identification. Once more, possibilities include URL encoding, and cookies. A central difference to note is that user identities persist, unlike sessions, which end after (usually) a relatively short period. An authenticated user identification token, such as may be encoded in a URL, will typically include not just the user identity but also an expiration time for the authentication. This way, a URL that is bookmarked and used at a later date will not automatically place the user in a logged in state. In setting the expiration time, once more we have the option of computing this with respect to the time of login or time of last access. If the expiration time is directly encoded in the URL, implementing the latter is hard – the best one can do is to compute expiration time from time of URL generation, and accept that different URLs will encode different expiration times. A more clever option is to encode in the URL not the expiration time, but rather a "login session identifier" in addition to the user identity. With this, information can be retained by the server regarding the current active authenticated sessions, and time of last activity can be used to close these sessions just as for ordinary sessions.

There will often be protected (not public) web pages that only authenticated users are allowed to access. It is straightforward to determine whether any particular request is being made in the context of an authenticated session. When an unauthenticated user attempts to access a protected page, the server could simply refuse the request, and return a "401: Unauthorized" HTTP response code. A substantially friendlier design is instead to return the login page, inviting the user to login to access the protected resource. Once the user successfully logs in, by submitting a form with the required user name and password, the user should immediately be sent the protected resource, rather than being sent a default logged in user home page. But the request for the protected resource occurred previously, as part of a previous HTTP transaction. To be able to supply this resource now, the web server has to remember the previous request. This is easily done by encoding this (state) information with the login page form, or by placing this information in a cookie. Sometimes, the login process may require multiple interactions, for example if the user has forgotten user name or password. Conceptually, there is nothing difficult about carrying state information regarding the original request through this sequence of requests. See Fig. 5. In practice, this seems to be an area where web programmers frequently make mistakes.

3 Cookies

Cookies are small files placed on the client machine by the server site, usually for the purpose of carrying state between client requests to the site. The precise location of these files depends on the browser and the operating system in use. With Internet Explorer on a Windows XP machine, cookies are in the folder C:Documents and Settings\ <username> \Cookies. HTTP requests have "relevant" cookies attached with them. These cookies can be read and processed at the server as it fulfills the HTTP request. See Fig. 6.

Cookies are so named in recollection of the well-known fairy tale of Hansel and Gretel. When these children have to go in the forest, they leave a trail of cookie crumbs along the way to remember the path they took. Cookies on the web similarly are used to remind the server of the context of a request – literally, where the user is coming from.

The client provides storage for the cookies, but has no control over the content of the cookie itself. The server writes the cookie to the client, but has no means to read a cookie at will – the client voluntarily sends relevant cookies at the time of an HTTP request. This delineation of responsibility creates an interesting interplay between two parties who need to cooperate to make the mechanism work, but at the same time have to make sure to limit what the other party is allowed to do, since there is the possibility that it will do something malicious.

From the client perspective, cookies take up storage resources and cookies can leak private client information. Considering storage, if the client were not to impose any limits, it is conceivable that a web site could use the client machine as free storage for anything it wishes to save. To prevent this, most browsers limit the size of a cookie and the number of cookies for one domain.

Typical storage limitations are 20 cookies per domain and 300 cookies total. 4 kilobyte size limit per cookie.

Furthermore, to prevent a web site from maliciously overwriting a competitor's cookie, most browsers limit the overwriting of a cookie to the domain associated with the cookie at the time it was set. This domain is required to be a suffix (super-domain) of the domain that originally set this cookie.

Example 2 www.widgets.com *can set a cookie with a domain of* widgets.com*. This cookie can subsequently be overwritten by* my.widgets.com*. If the cookie had its domain set as* www.widgets.com*, then it couldn't have been overwritten by* my.widgets.com*.*

Cookies may store private client information. The client has a measure of control over the dissemination of this information since cookies are not available for anyone to read: instead they are explicitly sent by the client with HTTP requests. The determination of which cookies are "relevant" to a particular request is made on the basis of the domain name and the path within the domain of the URL being requested. These are specified for each cookie. The specification at the cookie must match the prefix of the path and the suffix of the domain name requested for the cookie to be considered relevant. The client sends only relevant cookies to the server, and hence avoids cross-transmission of information across sites.

The path prefix notion is not interpreted at the granularity of links, but rather as a string prefix. Thus not only is catalog a prefix of catalog/page1.htm, but so also is catalog/page. We may use a path specification of catalog/page in a cookie, even though it is itself not a valid file-system path, if we wish to see the cookie with requests for pages of the catalog, but not for resources like catalog/index.htm or catalog/help.htm.

Example 3 *Let us look at what the cookies may actually look like in the example of Fig. 6. The first time the customer placed something in a shopping cart, the HTTP response includes the following line as part of the HTTP header:*

```
Set-Cookie: ShopperNum=7; path=/; expires=Monday, 09-Feb-10 11:12:20 GMT
```

The cookie identifies the requester as shopper number 7. Furthermore, the server has requested that this cookie be sent back to it with every request to it, with no restriction on the path to the resource.

The next time the user visits any page on the web site, whether to add something to the shopping cart, or just to browse, the HTTP request carries along with it in the header:

```
Cookie: ShopperNum=7
```

When the server receives this request, it can use this information in the cookie to find the session it belongs to, locate the corresponding data structure, and restore session state before serving the request.

Going further, beyond the Figure, suppose the user eventually gets to the checkout area, and the accounting department has its own procedures. The first request to that area of the web site may send back:

```
Set-Cookie: ShopperNum=23; Value=Low; path=/accounts
```

Note that there is a path restriction, so these new cookies are sent only when requests go to resources with the string accounts *in the beginning of the path. There is no expiration date set. So these cookies are supposed to expire at the end of the current session, which effectively means when the browser window is closed. Note also that we have chosen to name one cookie exactly the same as the previous one. We have done so purely to show, in this example, that it is possible to do. There is no requirement of uniqueness between cookies. The next time the client makes a request to the accounts portion of the web site, it sends:*

```
Cookie: ShopperNum=7; ShopperNum=23; Value=Low
```

When it makes requests to any other portion of this web site, it still sends:

```
Cookie: ShopperNum=7
```

From the server perspective, the primary worries are that the cookie may be deleted by the client (or not sent with a request with which it should have been), and that the cookie may be modified by the client, giving the server incorrect information. To prevent the latter, cookies should be encrypted. To ameliorate the former, a cookie should contain only minimal client and state identifying information, which can be used to access detailed information at the server, including client history, personal details, and so forth. This is desirable for several reasons: we want to keep cookies as small as possible, to minimize the burden on the client as well as the cost of transmission; we worry about loss of client privacy if a third party were to read the cookie (we do encrypt what we store in the cookie to foil casual eavesdropping, but still ...); we wish to support clients accessing our website from multiple machines, which do not share cookies.

Cookies as History

Notwithstanding the above, there frequently is good reason to store additional information in a cookie. A common situation concerns web sites that do not require user registration or authentication, yet would like to utilize the customers' history of interactions. For instance, mapquest stores addresses that have recently been queried in a cookie, making it possible to reuse such an address in a future search without having to type it in.

Conceptually, mechanisms to maintain state across HTTP requests, such as cookies, are valuable in providing greater functionality to the user. However, they also come with the downside that history can be remembered by the server – something that the user may not relish. The client, of course, has the option of deleting cookies, or disallowing cookies altogether. But short of these extreme options, there is no mechanism for the client to examine the contents of cookies and selectively manage this storage.

As long as each cookie stores history with respect to a single web site, this is not much of an issue: after all the web site could, and typically does, store all this information in its own servers. The only concern is one of user identification by means of the cookie – history may even be stored for non-members as discussed in the mapquest example above.

A much more problematic situation is when cookies can store history across web sites. Given the protocol, this appears not to be possible. However, see Fig. 7. Recall that downloading a single web page typically involves multiple HTTP requests on account of embedded images. There is nothing that requires these embedded images to be from the same web server: in fact, advertising is frequently served from a third party advertisement server. These web servers themselves could also use cookies, for instance to ensure no repetition by keeping a history of advertisements served recently. The advertisement server also knows precisely which web page at which web site it is serving an advertisement for. If multiple web sites use the same advertisement server, this server is able to collect user history at all these sites, and cross-correlate them.

Taking this idea further, there is no need for the embedded image actually to serve advertising or even pretend to do anything useful. A web page may embed an image not for display, but rather solely for the purpose of getting a URL request to a different site. Such an image is called a *single-pixel GIF* or a *web beacon*. In Chapter 16, we will see how non-cacheable web beacons can be used to get a more accurate count of page visits than would otherwise be possible.

Nothing compels a client to send a cookie with its HTTP request. Most browsers do permit the user to turn cookies off – neither accepting nor sending cookies. To deal with the problem of cross-information collection, many browsers now provide an intermediate option as well – cookies are only sent with (and accepted from) requests to URLs that were explicitly requested by the user, either by keying it in or clicking a link. Thus, with this setting, embedded images will be retrieved with cookies turned off.

Implementing Cookies

Cookies are set at the client using a Set-Cookie directive in the HTTP header. In addition to cookie name and contents, this directive also states the domain and path for the cookie, as well as other cookie properties such as expiry time and whether it may transmitted in the clear (unencrypted).

The HTTP syntax for setting cookies is
Set-Cookie: <name>=<value> [, <name>=<value>]* [; expires=<date>] [; path=<path>] [; domain=<domain-name>] [; secure] [; httponly]
In other words, a cookie is a list of one or more name-value pairs. The names are known to the client, the values are encrypted by the server, and stored by the client without interpretation.
All other attributes of Set-Cookie are optional – if they are absent, reasonable default choices are made. Note that the HTTP specification is silent with regard to cookies – the "standard" in this regard is really being driven by what the major browser vendors choose to implement. For instance, the httponly flag was not present in the original Netscape definition, but has since been introduced by Microsoft and is used widely.
Here is an example of the use of Set Cookie: Set-cookie: mycookie=myval; domain=mycompany.com; path=/; expires=Fri, 08-Dec-06 14:23:00 GMT
The name of the only variable this cookie defines is mycookie, and this variable is set to a value of myval. A cookie could define multiple variables. The expiration date and time is typically defined in GMT, since the local time at the web server may be different from that at the web browser, but each should know how their local time relates to GMT.

The browser includes the cookie in its HTTP request to the specified domain and path, through the environment variable HTTP_COOKIE, included in the HTTP request header. The value of this variable is a semicolon delimited list of cookie variable name/value pairs for the domain and path in question. The web server can parse this string to obtain the cookie information it needs.

There are a few tricky issues worth discussing in this regard. First, we have thus far seen server side applications that access HTTP headers, but the generated result is just the content file. The response HTTP headers have all been generated by the web server, and not the application. Clearly, this is not sufficient in the case of setting cookies, since the web server doesn't know what to put in it. This requires that server side languages, in addition to result page generation, provide facilities for setting HTTP headers. A common solution, followed by languages such as PHP, is to permit header information to be set provided this is done before any content is output. For instance, PHP has a setcookie() PHP command that mimics the Set-Cookie HTTP directive. With respect to cookie retrieval there are no issues: PHP can read the HTTP_COOKIE environment variable, parse out the name value pairs from it, and, for each name value pair, initialize one variable named name with the value value.

Since each set cookie has a performance cost, servers should consolidate multiple variable of interest into one single cookie in most cases. Some times, it may be worthwhile to break this up into a few groups that tend to be set together. E.g. variables that are rarely changed after initial set-up can be in one cookie, whereas operational variables updated on each access may be in a different cookie.

The security flag asks the client not to disclose the cookie to anyone except over a secure connection, such as SSL. If this flag is set, the cookie will not be available when insecure HTTP requests are made. Even if this flag is set, the cookie should still be encrypted, to prevent it from being tampered with by the client: secure links only provide protection against third parties. In particular, if a cookie, rather than server side state information, records that a user is logged in, such a cookie must only be sent over a secure link to avoid replay attacks.

Cross site scripting is an attack where the attacker page runs a script on the client machine to glean information about other sites through cookies. The httponly flag indicates that the cookie will never be accessed by a script, and can be used to foil such cross site scripting attacks.

An expiration time specified for the cookie indicates for how long the client is requested to keep the cookie. As with so many other things on the web, this is a request, which the client may choose to honor at its pleasure. More importantly, we have potential issues with time synchronization. There has been considerable research on effective clock synchronization in distributed systems, and all those challenges are certainly present on the web. But, even beyond these, we have the additional possibilities of a client machine having its date erroneously set to something completely bogus. In short, the server should manage expiration of sessions itself, with its own timers, and suitably set cookies. The cookie expiration time is merely a hint being provided to the client to assist it in managing cookies storage.

Though we have described the mechanics of cookies and URL extension in considerable detail above, much of this may be hidden from the typical web programmer by the server side scripting language used. Most such languages, including PHP, provide session facilities in the language. The programmer can use these as provided, and have the necessary cookies taken care of under the covers.

4 Personalization

Once a user is identified, whether authenticated or not, this identification can be used to provide the user with a personalized experience. In this section, we will look at what personalization means.

There are two, quite distinct, aspects to personalization. The first is *customization* – where the user explicitly specifies parameters and expects these to be reflected upon subsequent access. For instance my.yahoo.com asks the user to specify the features of interest as well as the layout they desire. Users will take the time to customize their interaction with sites that they visit frequently, if given the opportunity. From the server standpoint, it is straightforward to store these preferences in a user profile and to apply these automatically to each response as soon as the user is identified.

A second aspect to personalization is known as *individualization*. This occurs where the user does not explicitly do anything, but the server tailors the presentation to known user circumstances or observed user behavior. For example, Google serves advertising with the results of a search based upon the terms that the user searched for. Other, cruder, possibilities are to try to use the IP address, and any other available information, to categorize the user, and then serve say a geographically individualized page.

Group-tailoring is an important variant of individualization. Rather than tailoring the presentation to a single user's observed behavior and preferences, the server can associate the user with a group, and then use a larger basis on which to figure out how to tailor content for the user. For instance, Amazon uses similarity of purchase histories to suggest books to individual users. See discussion on collaborative filtering in Chapter refchap:collab.

5 Putting It All Together

To put all the concepts above together, we discuss here the architecture of a "Single Sign-On" system, where a central authority, such as Microsoft Passport, is used to get a user logged in to multiple sites. When a site participating in this authentication "group" gets a request, it requires that the request be authenticated by the group's central authentication server – Microsoft Passport in our example.

There are a few challenges in making this happen properly. The first time a user attempts to access a protected page, it is straightforward to redirect the client to the authentication server. Once authentication is successfully completed, how we get the user back to where they were is the first challenge. We need a user to have to log in just once and access multiple sites in the group. How to convey the logged in status securely to a site is a second challenge, given that cookies are domain specific. A third challenge is to log the user out of all participating sites if the user chooses to log out of one of them, to prevent unauthorized access, for instance by another user at a public terminal.

Let us start from the beginning, and look at the simple things first. A user requests a protected page from site A. The site finds that the request is not authenticated, and so serves up a page with a sign-in link. When this link is clicked, the user is redirected to the authentication server, which serves up the login page over a secure connection. The authentication server also receives the URL of the protected page originally requested. The user fills in the requisite information (typically a user id and password), and submits this form to the authentication server with an HTTP POST. The authentication server verifies the user credentials. Once authentication is successful, it does two things. One, it sets a cookie so that the next time the user visits the authentication server (within some timeout period) there is no need to furnish credentials again. In this cookie, it also records the list of all logged sites visited, at the current time this is just site A. Two, it redirects the user back to the protected page on site A the user had originally requested, and whose URL it had received with the original redirect. Furthermore, it encodes a user authentication "ticket" in the request URL so that site A now recognizes the request as coming from an authenticated user. Site A now serves up the requested page, and also sets its own cookie at the client so that future requests to site A do not require URL encoding.

Now suppose that this logged in user attempts to visit a page on a different site B within the same authentication group. Site B has no evidence that the user is authenticated. So it behaves in exactly the same way as site

A above – serving up a page with a sign in link. This time, when the user clicks this link, the authentication server receives its cookie, which it can update, noting site B as also visited. The authentication server does not ask the user for any credentials, instead it redirects the user back to the appropriate page on site B with a ticket encoded in the URL. From here on, everything is just as in the case above.

Finally, after a while, suppose the user clicks on a logout link at site A. This link causes a logout request being sent to the authentication server, which reads the cookie to determine all the sites visited – sites A and B in this example. It updates the cookies to clear the list of visited sites, and to set the status of the user to logged out. It then serves up a logout page that includes logout "images" from each site visited, A and B. When the client loads these images, a request is sent to each site, enabling it to clear its cookies. Optionally, this logout page can redirect to the "official" sign out page at site A once this sequence of logouts is all completed. See Figure 8.