DPSP™ – Dynamic PL/SQL™ Server Pages

Introducing Dynamic PSP Version 2

Version 2.1.3

Copyright© 2000-2002 by N-Networks
All rights reserved.

Document ID: 009-200002-005
Last Revision: October 22, 2002

COPYRIGHT INFORMATION AND ACKNOWLEDGEMENTS

DPSP, Dynamic PSP, OPSP and Objective PSP are trademarks of N-Networks

DPSP Interpreter, DPSP System Objects and this document are Copyright© 2000-2002 by N-Networks

Oracle is a registered trademark of Oracle Corporation.

PL/SQL, Oracle8*i*, Oracle9*i*, Oracle Internet Server, Oracle WebServer and Oracle WebServer Option are trademarks of Oracle Corporation.

Sun, Sun Microsystems, the Sun Logo and Java are trademarks or registered trademarks of Sun Microsystems Inc. in United States and other countries.

Other company or product names are mentioned for identification purposes only and may be service marks, trademarks, or registered trademarks of their respective owners.

Although every effort was taken to make this document as accurate and complete as possible, no guarantees whatsoever are given in regard to document's accuracy and completeness. Also, no guarantees are given that this document fully covers the functionality of the product it describes.

Information in this document is subject to change without notice.

INTRODUCTION

WHAT IS DPSP AND WHY WE DECIDED TO CREATE IT?

DPSP™, or Dynamic PSP™, is PL/SQL™ Server Pages interpreter and compiler designed for Oracle8*i*™ RDBMS and Oracle® Application Server (OAS)/Oracle9*i*™ Application Server (iAS), a simple yet very powerful server-side scripting solution for Oracle AS. It is installed into Oracle8*i* RDBMS as several PL/SQL packages and Java™ classes and is instantly available after that, provided that OAS/iAS OWA packages are already installed and target schema is published via iAS. DPSP units are created, edited and managed via web interface requiring no new software installation for developers beyond any HTML4.0/DHTML-compliant browser. Optional Java-based Unit Editor is also provided with enhanced user interface.

Although Oracle has a term 'PSP' with the same meaning of 'PL/SQL Server Pages', it differs seriously from what you will read about in this document. After a bit of playing with Oracle's PSP we decided that its way is too limited in functionality we needed, so we decided to create our own server-side extension to Oracle RDBMS/AS and devised it *Dynamic PSP* (DPSP) as this name best describes what we intend to achieve but sets us aside from Oracle's PSP approach. We believe Dynamic PSP is a viable alternative to Oracle's PSP and JSP as well, because it doesn't require experienced Oracle PL/SQL programmers to learn Java and doesn't limit them in the way they can get things done.

DPSP shares common syntax with Oracle's PSP, but extends it with some helpful features, like a number of predefined functions, dynamic execution (unlike Oracle's one-time compilation approach), unlimited number of parameters to any DPSP unit (in PSP, the number of parameters is fixed), and more, while preserving syntax and common functionality of PSP.

DPSP VERSION 2

DPSP Version 2 (DPSP2) is the next stage in evolution of Dynamic PSP. It shares the syntax and basic concepts with previous version, but has different internal architecture and implements a number of enhancements to the original Dynamic PSP v1.

NECESSARY READER'S BACKGROUND

Readers of this document are assumed to have some knowledge of xSP (any server pages) in general, extensive knowledge of Oracle8*i* and PL/SQL and good understanding of Internet technologies (WWW, HTTP, HTML, scripting languages, etc.) We will provide you with some explanations of more complex topics throughout the document, but above is a must. Experience with Oracle9*i* Internet Application Server™ (iAS) and Oracle PSP is a plus, but generally not required, as we will describe DPSP syntax and provide some insight into iAS in this document. We recommend browsing through official Oracle8*i* documentation and Oracle Internet Application Server documentation for all the questions regarding Oracle products. All Oracle documentation can be obtained from Oracle Technology Network (OTN) at http://technet.oracle.com/

Copyright© 2000-2002 by N-Networks. All rights reserved.

DPSP BASICS.

If you are already familiar with Dynamic PSP v1, you can skip this chapter and read the following chapters: "*Differences Between Dynamic PSP v1 and Dynamic PSP v2*" and "*New Features*".

This chapter describes basic syntax and features of Dynamic PSP using 'learn by example' approach. We will go through a series of examples, from simplest classic 'Hello World' to more complex dynamic table printing and formatting to see DPSP in action. We assume that you are already familiar with HTML and server-side preprocessing concepts.

The examples mentioned in this document can be found in the standard distribution kit of DPSP2. These examples are referenced here by a *unit identifier* as 'unit 1', 'unit 2', etc.

BEFORE WE START.

Before we start with examples, let's define the concept of server pages. A server page is a template with embedded code written in a particular programming language that fills the template with actual data. All this processing is performed on the server before returning the final document to the requesting client. The main difference between server pages and CGI (Common Gateway Interface) programs is that CGI deals with different programs that return content, while server pages are document templates that are processed by single program (server pages processor). Server pages usually share the same syntax for defining program blocks that will be processed on the server, though used programming languages themselves may be very different. For example, ASP uses Visual Basic as programming language, JSP uses Java, and PHP uses its own C-like language.

Dynamic PSP uses Oracle PL/SQL language for dynamic content generation. The following simple examples will show the basic syntax of a PSP page.

EXAMPLE 1. HELLO WORLD FROM DPSP.

Let's start with usual Hello World application (see unit 1):

```
<html>
<head>
<title>DPSP Example 1</title>
</head>
<body>
<% print('<b>Hello World!</b>'); %>
</body>
</html>
```

As a result of execution there will be a simple static page with **"Hello World!"** being the only text on it. Not much of dynamic content here, but you have already got the idea, don't you? Now let us try a more complex example.

Copyright© 2000-2002 by N-Networks. All rights reserved.

EXAMPLE 2. OUTPUTTING A TABLE.

Let's create a simple dynamic web page, dynamic in the sense of its content, not behavior, i.e. we're going to retrieve some data from the database and present it to the viewer of the web page – that's what all server-side scripting technologies are about (see unit 2):

```
<html>
<head>
<title>DPSP Example 2</title>
<body>
<table width="100%">
<% for r in (select * from all_users) loop %>
  <tr>
    <td> <%= r.username %>  </td>
    <td> <%= r.user_id %>   </td>
    <td> <%= r.created %>   </td>
  </tr>
<% end loop; %>
</table>
</body>
</html>
```

Now let's see what we've just done here.

Actually we defined a block of PL/SQL code that will be interpreted by a runtime interpreter and all **<%...%>** tags will be replaced with the execution results. You may notice two different kinds of **<%...%>** tags here: plain **<%...%>** and **<%=... %>**. The first form defines a **fraction** or **block of PL/SQL code**, the second form will evaluate **PL/SQL expression** within the brackets and replace the tag with the result of the evaluation. In plain English, the above code says:
Select all rows from `ALL_USERS` table or view and output them in a loop. For each selected row open <tr> element, then output three <td> elements with contents of `USERNAME`, `USER_ID` and `CREATED` columns of current row and then close the <tr> tag. As you may note, the case is not important here – Oracle is case-insensitive by nature.

Copyright© 2000-2002 by N-Networks. All rights reserved.

EXAMPLE 3. FORMATTING THE TABLE.

Ok, we managed to output the table, but what about formatting its presentation dynamically? For example, how can we make it look striped (that is, apply different formatting on odd and even rows of the table)? Let's use the code from example 2 and enhance it with CSS styles (see unit 3):

```
<html>
<head>
<title>DPSP Example 3</title>
<style type="text/css">
<!--
.oddrow {background-color: #EEEEEE;}
.evenrow {background-color: #CCCCCC;}
-->
</style>
</head>
<body>
<table width="100%">
<%! rcls varchar2(10); %>
<% for r in (select rownum, a.* from all_users a) loop
    if mod(r.rownum,2)=0 then
        rcls := 'evenrow';
    else
        rcls := 'oddrow';
    end if;
%>
  <tr class="<%= rcls %>">
    <td> <%= r.username %> </td>
    <td> <%= r.user_id %>  </td>
    <td> <%= r.created %>  </td>
  </tr>
<% end loop; %>
</table>
</body>
</html>
```

Now the even table rows have a color different from the odd rows. Here you see a new tag - **<%!… %>** - which allows declaring a global variable and use it in other statements throughout the DPSP unit. Here we define a string variable `rcls`, and assign two different values to it that are then output with **<%= …%>** tag within another HTML tag. We also slightly modified the `select` statement to include `rownum` into it, which is not available by default. `rownum` is used to determine if a current row is odd or even. Note, that you can use DPSP tags *within* HTML tags, for example for dynamically setting tag attributes. You cannot use them within other DPSP tags though. In rare cases when you need to print DPSP tag delimiters from within DPSP (for example, to print out DPSP source code), you can 'escape' them by using the following syntax:

```
<\% and/or %\>
```

In this case, backslash character will be stripped from the output, but remaining characters will be output as they are and will not be interpreted as DPSP tag start/stop sequences.

These three examples should have given you enough information to start with. Next chapter will discuss DPSP syntax and architecture in more detail.

Copyright© 2000-2002 by N-Networks. All rights reserved.

DPSP SYNTAX AND INVOCATION.

ORACLE PSP PAGES AND PACKAGES VS. DPSP UNITS.

Oracle PSP and DPSP both employ Oracle PL/SQL as server-side programming language for retrieving or modifying data in the database, generating page content and do a lot of other things on server before returning page to the requesting client. There are some differences between the two though.

In Oracle PSP you write a dynamic page as a text file and then use a special utility that parses your page, creates a stored procedure source code and loads/compiles this stored procedure into Oracle RDBMS. Each external parameter you define in PSP page becomes a parameter of the stored procedure. This limits number and names of external parameters for this particular PSP page. Each time a change in functionality of the page is required, you need to edit the source file (.psp file) and reload the page into Oracle, which in turn triggers code regeneration and recompilation. Any syntax error during this process terminates loading and you need to fix it before PSP page can be loaded into Oracle.

In contrary, each DPSP block is stored in special table in the database, has *unique numeric identifier* (ID), and is parsed and compiled dynamically when accessed. These blocks are called DPSP units. DPSP unit can represent complete page or a logical block (subroutine) that can be called from other DPSP units allowing you to reuse the code. DPSP units can be called from each other by ID, and can accept unlimited number of external named parameters. DPSP units are dynamically parsed and compiled and all changes you made to them become instantly available as soon as you save edited object. Any syntax errors will be reported when unit is executed for the first time or compiled. DPSP units can be treated as procedures that output directly to the client or as functions that return result in a string that can then be modified before returning it to client. Both forms are automatically available with no changes to the unit code.

When new unit is created, it is assigned a unique numeric ID, and a name. Although name of the unit can be changed later on, ID cannot be changed and will always identify the same unit once it is created.

Negative unit IDs are reserved for system units and generally should not be used for your custom units.

DPSP units can be compiled into packages, for example, when you want to protect them from further modification or source code assessment (in this case, you can extract unit package source after compilation and obfuscate it using WRAP utility). Also, compilation is required when recommended secure setup procedure is followed to make unit 'published' and available at the website. When DPSP unit is debugged and complete, you set several attributes to enable its compilation and compile it into a package. DPSP interpreter checks if a package exists and executes it if found and the package is valid, otherwise it retrieves the unit source code and interprets it.

Copyright© 2000-2002 by N-Networks. All rights reserved.

GENERIC DPSP PAGE OUTLINE AND DPSP TAGS.

Generic DPSP page includes standard static HTML text plus dynamic content, which can consist of data retrieved from the database or submitted by a visitor through a form, blocks of HTML that are displayed conditionally, etc.

Special tags are used to deliver the dynamic content. Definitions of DPSP tags are as follows:

<% PL/SQL statement; … %>

This tag surrounds a block or fraction of PL/SQL code. All such fractions are combined into an anonymous block or compiled into a PL/SQL package (the way a unit is processed is determined by its attributes). The code should follow all PL/SQL requirements, restrictions and syntax. You can access any standard and custom Oracle objects (procedures, tables, views, etc.) in these blocks as well as some predefined DPSP functions. Oracle database security controls access to all Oracle objects, like tables, views or packages, the same way as for any other SQL or PL/SQL code, that is DPSP unit gains only rights of the database user that executes it.

Example:

```
<% for r in (select * from all_users) loop %>
  any HTML/DPSP to put in a loop here
<% end loop; %>
<% if someVariable = someValue then %>
  This will only be seen if someVariable = someValue
<% else %>
  This will only be seen if someVariable <> someValue
<% end if; %>
```

In the first example, we do a **SELECT** from **ALL_USERS** view and the calling user should have a SELECT permission on that view. The second example demonstrates conditional display of static HTML. Note that each complete PL/SQL statement should be terminated with semicolon (for example, if … else … end if;).

<%= PL/SQL expression %>

This tag surrounds an expression. The result of the expression is outputted in the place of the tag. Note that the expression should NOT be terminated with semicolon, unlike PL/SQL statements in other types of DPSP tags. Any valid PL/SQL expression is allowed in this tag.

Example:

```
Current date is <%= sysdate %>.
```

<%! PL/SQL variable declaration; … %>

This tag surrounds a global variable definition block. You can combine several variable definitions in one block. Generally, you should follow conventions for Oracle DECLARE block of an anonymous PL/SQL block. The tag can be issued anywhere in the DPSP unit as many times as you want, provided that you do not redeclare the same variable. All declarations will be combined into a single DECLARE block before compilation/execution. You should not issue **DECLARE** keyword within this element. All variables declared within these tags become GLOBAL, that is they will be visible throughout the whole unit, unlike inline, or local **DECLARE**s which are visible only within a subsequent **BEGIN..END;** block.

Copyright© 2000-2002 by N-Networks. All rights reserved.

Example:

```
<!-- global declarations -->
<%!
  Var1 VarChar2(30);
  Var2 Number := 0;
%>
<!-- local declaration -->
<%
declare
  someVar number;
begin
  some code – someVar is visible here;
end;
%>
<!-- someVar will not be visible here -->
```

(See Example 3 above for the usage of all these tags in a complete dynamic page.)

<%@ PSP preprocessor directive %>

This tag is used by PPSP (Procedural PSP) preprocessor to define some attributes of the unit, which affect unit compilation. Recognized directives are:

`procedure <procedure_name>[(parameter spec [, parameter spec[, …]])]`

This directive defines an internal procedure (that is, a procedure which is visible only to the unit) with some optional parameters. You can define as much procedures as you want. Order of declarations is only important if you want to call one defined procedure from the other – in this case the called procedure must be declared before calling procedure. Global variables declared with `<%! %>` tag are visible within these procedures, but variables declared in procedures are only visible within procedure body.

`main`

If `procedure` directive is used, `main` directive defines the main unit body. It is mandatory to issue `<%@main%>` to identify main body if `procedure` directive is used anywhere in the unit code. You can issue this directive before or after `procedure` directives, preprocessor will put the code in right order automatically.

Please note that Oracle PSP uses this tag for slightly different purposes – for defining page parameters and various other attributes. Oracle PSP directives are currently ignored by DPSP and PPSP preprocessors.

Example:

```
<%! num integer := 0; %>
<%@procedure onerow(nm in varchar2)%>
 <%! localVar integer := 0; -- this variable will only be visible in onerow()%>
  <% num := num + 1; %>
  <tr>
    <td align="center"> <%=to_char(num)%> </td>
    <td align="right"> <%=nm%> <%='&'%>nbsp; </td>
    <td> "<%=owa_util.get_cgi_env(nm)%>" </td>
  </tr>
<%@main%>
<table border="1" cellpadding="1" cellspacing="1" cols="3"
       style="font-size:10pt" width="100%">
  <caption align="left" style="color:blue;font-size:12pt;font-weight:bold">
    CGI environment variables:
```

Copyright© 2000-2002 by N-Networks. All rights reserved.

```
  </caption>
<% -- do the printing using our procedure
  onerow('GATEWAY_INTERFACE');
  onerow('REMOTE_HOST');
  onerow('REMOTE_ADDR');
  . . .
%>
</table>
```

Copyright© 2000-2002 by N-Networks. All rights reserved.

EXECUTING DPSP UNITS.

Each DPSP unit can be executed from the web. DPSP units can represent dynamic pages or reusable parts of a page, forms, form handlers, etc. To execute a DPSP unit you need to make an HTTP request from any browser to the DPSP server.

URL format is as follows:

[<proto>]server[:<port>]/<DPSP_schema>/!go?id_=<unit_id>**[&param1=value1&param2=value2…]** *

where
<proto> is optional protocol specifier (for example, http://)
<port> is optional server port number (default is 80)
<DPSP_schema> is DPSP schema access prefix (as per Oracle PL/SQL Gateway specification)
<unit_id> is DPSP unit identifier (numeric)

You can pass unlimited number of the named parameters additional to the unit. If you specify several values for one parameter name (e.g. 'param=value1&param=value2'), they will be passed to the unit as an array. Be sure NOT to name any of your custom parameter **id_**, as this name is reserved for a mandatory unit identifier parameter. **!go** is also a mandatory member of a DPSP unit call URL – it is the DPSP call hub procedure which initially accepts all parameters, processes them and then calls the DPSP kernel to process the request. Exclamation sign in front of GO tells the 9iAS server that the hub procedure follows 'flexible parameter passing' conventions. If you designate a DPSP unit as FORM handler, be sure to specify ACTION="!go" and declare **id_** as INPUT TYPE="HIDDEN" parameter or append it to **!go** in ACTION attribute of the FORM:

```
ACTION="!go?id_=unit_id%>"
```

You can also use unit name attribute in place of unit_id (id_=<unit name>), but you should take care about unit name uniqueness in this case – the system does not enforce uniqueness of attributes. The system will fail to resolve the name to id if several units have the same name attribute value.

* Although this is mandatory call layout for DPSP, one can use Apache's mod_rewriter module for hiding this layout. For example, mod_rewriter may be used to translate URLs like /dpsp_schema/unit_id to /dpsp_schema/!go?id_=unit_id automatically.

Copyright© 2000-2002 by N-Networks. All rights reserved.

DIFFERENCES BETWEEN DYNAMIC PSP 1 AND DYNAMIC PSP 2

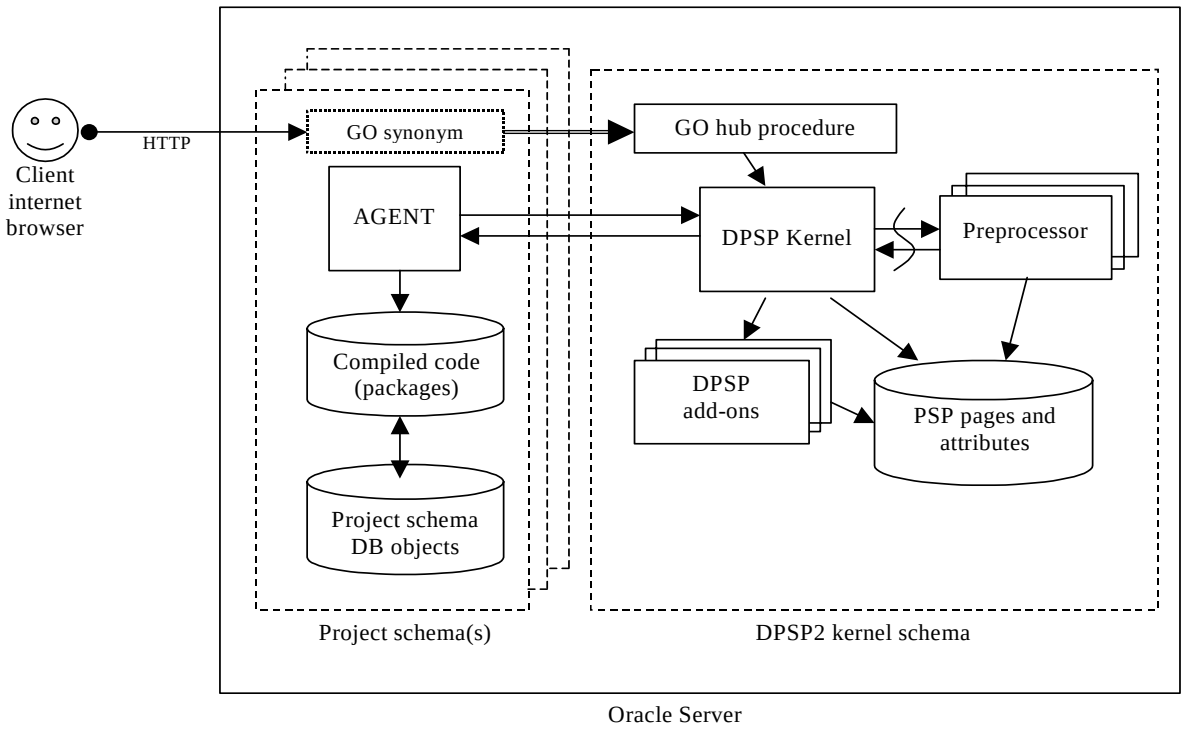
This chapter briefly goes over differences in architecture, API and built-in functionality between Dynamic PSP v1 and Dynamic PSP v2.

NEW INTERNAL ARCHITECTURE

DPSP2 is built around the concept different than that of DPSP v1. DPSP2 core is broken into modules, all of which provide different functionality and depend on DPSP Kernel. DPSP2 introduces internal persistent hierarchical data storage, similar to Windows Registry, which is called DPSP Registry and is independent from the rest of the DPSP core. All DPSP modules store their properties and other data in the Registry. DPSP Registry also exposes an API that allows developers to store and manipulate their own data using this hierarchical storage. DPSP2 is built as a set of modules interacting with each other at runtime to achieve needed functionality. For example, when a unit is executed, the call hub procedure calls the Kernel, which in turn calls the global Preprocessor module. Preprocessor module generates executable code from the page source depending on the unit type, and passes it to Kernel for execution. The Kernel sets up the page environment, including parameters passed to the page, and calls project schema-resident Agent module, which executes the page code in project schema context. Kernel then retrieves the result of code execution and passes it over to the HTTP gateway, which returns the page to the requesting client. This modular approach allows extending DPSP functionality by adding new modules to it (preprocessors for different languages, postprocessors for additional output tweaking/processing, etc).

DYNAMIC PSP VERSION 2 CALL DIAGRAM

The call flow is presented on the following diagram:



When a user makes an HTTP request, PL/SQL Gateway invokes the GO procedure in context of the DPSP project through synonym. The GO hub procedure processes the call parameters and passes them to the Kernel. Kernel retrieves the unit code and passes it to the PSP Preprocessor. PSP Preprocessor determines if the code should be regenerated or there is a compiled version of the unit available and generates the executable code for the unit if needed. Kernel then calls project schema Agent and requests it to execute the unit in context of the project schema. Dynamic or compiled unit code is executed against the project schema

Copyright© 2000-2002 by N-Networks. All rights reserved.

(accessing all schema objects as needed). The result of execution is stored in internal Kernel result buffer and is returned to the requesting user via HTTP gateway (it may be post-processed by a postprocessor module before it is returned if needed – for example, if the result document is XML and an XSLT template should be applied to it). If the unit requests other units to be executed, these calls are carried out by the Kernel and results of calls are either embedded into result buffer or passed to the calling unit for further processing.

Copyright© 2000-2002 by N-Networks. All rights reserved.

NEW FEATURES

- **Extensibility**: Dynamic PSP v2 is designed with extensibility in mind. DPSP2 can be easily extended with new features without the need to change the internal architecture of the kernel thanks to modular architecture.
- **Modularity**: DPSP2 consists of several components (modules) and each module is installed separately. Some modules are required and some are optional. New modules can always be added to the existing set, seamlessly extending the capabilities of the system.
- Enhanced **built-in API** that covers all aspects of web-based development and provides complete control over HTTP output your application generates.
- **DPSP Registry** subsystem provides hierarchical persistent data storage facility similar to Windows Registry. This subsystem is extensively used by DPSP2 modules, but can be used standalone with any PL/SQL applications.
- DPSP2 has strictly defined **kernel** separated from DPSP2 projects. The kernel should exist as single instance in the database, and can be used to support unlimited number of independent DPSP projects in different schemas. All DPSP projects use shared kernel services, but operate in their own database schemas.
- **Independent Preprocessor modules** perform DPSP2 unit code processing. Each DPSP2 unit has attributes defining which preprocessor to use to generate the executable page code. DPSP2 system can have unlimited number of specialized preprocessors. FLAT, DPSP and PPSP preprocessors come standard with DPSP2. FLAT preprocessor does not do any processing and is used for static HTML/XML/JavaScript units, DPSP preprocessor implements DPSP v1 syntax and features and PPSP preprocessor implements Procedural PSP extensions to DPSP. Optional OPSP (Objective PSP) preprocessor is also available.
- DPSP2 modules interact with each other using **Dynamic SQL**. Thus changing, upgrading or removing one module does not cause automatic cascaded invalidation of dependent modules.
- DPSP2 widely uses **synonyms**, which minimizes the risk of naming conflicts and allows effective code sharing.
- DPSP2 implements new system for **unit attributes** which allows assigning any unit any number of attributes without the need to change internal tables layout. This new system allows assigning new attributes to existing units to implement new functionality in the future.
- Centralized **exception handling** which provides a number of options for handling errors in DPSP units, including printout of the source code with error source line/block highlighted.
- Strict internal **security**. DPSP2 implements strict access control for DPSP units. All operations with DPSP units except execution (creating, editing, etc.) require explicit user registration in DPSP kernel.
- **Automatic versioning and archiving** of DPSP units. Kernel automatically saves backup copies of edited units and any unit can be reverted to its previous state if needed.
- **Automatic execution tracing** for DPSP units. Trace data can be used to gather unit execution statistics (for example, to detect unused units).
- **Session context subsystem** built into DPSP2 kernel allows storing session context from call to call eliminating the need to store this data on client as cookies or pass it as parameters from call to call. Each context gets its own unique identifier tied to session identifier (sessid). Session contexts can inherit some data from other contexts.
- **Extensive Internationalization Support** is provided in form of additional module (NLP, National Language Processing). This module allows creating international applications and translating user interfaces by supporting an extensible dictionary of translations. Units should not output any text to be translated directly, but rather through the translation engine, which will lookup the dictionary and

Copyright© 2000-2002 by N-Networks. All rights reserved.

output correct translated text if it can locate it in the dictionary. This module thus acts as a string resource manager.

- **Logical Names subsystem** implemented as additional module (Resolver). When installed, this module allows to assign each unit a logical name and to access DPSP2 units by their logical names rather than by their IDs. Logical names are unique and may be reassigned to different units allowing transparent unit exchange.

Copyright© 2000-2002 by N-Networks. All rights reserved.