Title
Implementation of Advanced Techniques for Automated Freeway Incident Detection

Authors
Abdulhai, Baher
Ritchie, Stephen G.
Iyer, Mahadevan

Publication Date
1999-12-01

CALIFORNIA PATH PROGRAM
INSTITUTE OF TRANSPORTATION STUDIES
UNIVERSITY OF CALIFORNIA, BERKELEY

Implementation of Advanced Techniques for Automated Freeway Incident Detection

Baher Abdulhai, Stephen G. Ritchie, Mahadevan Iyer

California PATH Research Report
UCB-ITS-PRR-99-42

This work was performed as part of the California PATH Program of the University of California, in cooperation with the State of California Business, Transportation, and Housing Agency, Department of Transportation; and the United States Department of Transportation, Federal Highway Administration.

Report for MOU 358

December 1999

ISSN 1055-1425

Implementation of Advanced Techniques for Automated Freeway Incident Detection

Baher Abdulhai
Assistant Professor and Manager of ITS Research
Department of Civil Engineering
University of Toronto
Toronto, Ontario, Canada M5S 1A4
Tel: (416) 946-5036, Fax: (416) 978-5054, baher@ecf.utoronto.ca

Stephen G. Ritchie
Professor and Chair
Department of Civil and Environmental Engineering
University of California, EG E4130
Irvine, CA 92697-2175
Tel: (714) 824-5333, Fax: (714) 824-2117, sritchie@uci.edu

Mahadevan Iyer
Ph.D. Graduate Student
Institute of Transportation Studies
University of California Irvine

PATH

AUGUST 17, 1999

EXECUTIVE SUMMARY

A significant body of research on advanced techniques for automated freeway incident detection has been conducted at the University of California, Irvine (UCI). Such advanced pattern recognition techniques as artificial neural networks (ANNs) have been thoroughly investigated and their potential superiority to other techniques has been demonstrated. Of the investigated ANN architectures, two have shown the best potential for real-time implementation: namely, the Probabilistic Neural Network (PNN), (Abdulhai and Ritchie 1997), and the Multi-Layer-Feed-Forward Neural Network (MLF), (Cheu and Ritchie 1995).

This project extended existing freeway incident detection research conducted under both PATH and under the ATMS Testbed Research Program, to operationalizes its principal findings. The most prosmising neural network, the PNN, was integrated into the UCI testbed for on line operation on the testbed network in Southern California.

The PNN incident detection system was re-coded in Java, to facilitate network communications and platform-independent operation. A Java-based graphical user interface has been developed. The GUI components include a display of the probabilistic neural network (PNN), the current input to the PNN, a sliding window display of the output (the computed incident probability every time step) and a sliding button to allow the user to specify the desired misclassification cost ratio. The GUI code is in the form of a Java Applet object and has a modular structure that makes it easier to incorporate possible future modifications and extensions. The PNN algorithm itself was then translated from C to Java as a stand alone application object and was interfaced to the GUI applet running on the same host. The GUI display is updated each time a new output is computed by the PNN. The PNN algorithm and the GUI display update run as separate threads of control in Java; this concurrency leads to better utilization of CPU resources. A new module for computing the principal component transformation of the volume and occupancy inputs was developed to replace using statistical packages for this transformation. This was needed for maximum portability and independence of the overall system. Another module for computing volume and occupancy historical Averages for different Times and Locations (ATLs.) was also

developed to prepare the ATLs from real freeway data. The PNN and GUI were tested and correct operation was confirmed with sample inputs from data files.

The whole package was interfaced to a remote C++ CORBA client program that acquires online CalTrans traffic data from a CORBA server in the Testbed. Communication modules were added to the CORBA client program as well as the PNN to enable online volume and occupancy data from different freeway sections to be sent from the CORBA client to the PNN at a specific rate (once every 30s). The data are sent to the PNN using a reliable TCP/IP streams sockets connection.

An on-line retraining module was developed as well. This module enables the TMC operator to initiate retraining on recently captured incident data, on-line without disturbing the operation of the system.

The PNN was then started on-line on a 5 mile section of the 405 freeway, for on line monitoring and testing. The overall on-line operation of the PNN was demonstrated to Caltrans engineers from D12. Currently, efforts are in progress to expand the network coverage to enhance the odds of capturing incidents. On line evaluation will be performed next.

TABLE OF CONTENTS

INTRODUCTION

Considerable research has focused on improving techniques for managing traffic in a real-time environment. From this perspective, proper incident detection and management are recognized to be key components of any potentially successful Advanced Traffic Management System (ATMS). Such detection has become a vital link in decision support for TMC personnel responsible for large and complex transportation networks managed by an ATMS.

The performance of conventional approaches to automatic incident detection has proven inadequate for every-day use at Traffic Management Centers (TMCs). Inadequacy stems from three main sources: (1) the less-than-perfect performance at the original site that the algorithm was developed for, (2) the lack of transferability to any new site, and (3) the inability of the algorithm to consider such important but often neglected issues as the prior (predicted) probability of occurrence of incidents, posterior probability of an incident after an alarm, and the unequal costs of misclassifying a traffic pattern as an incident or non-incident. A significant body of research on advanced solutions to these issues has been conducted at the University of California, Irvine (UCI). Such advanced pattern recognition techniques as artificial neural networks (ANNs) have been thoroughly investigated and their potential superiority to other techniques has been demonstrated. Of the investigated ANN architectures, two have shown the best potential for real-time implementation: namely, the Probabilistic Neural Network (PNN), (Abdulhai and Ritchie 1997), and the Multi-Layer-Feed-Forward Neural Network (MLF), (Cheu and Ritchie 1995).

This project extends existing freeway incident detection research conducted under both PATH and under the ATMS Testbed Research Program, and operationalizes its principal findings. The PNN will undergo extensive testing in UCI testbed as well as in real TMC environments. The final product of this effort is expected to be an operational neural-network-based freeway incident detection framework.

ARTIFICIAL NEURAL NETWORKS (ANNS)

Artificial neural networks, from the field of Artificial Intelligence, are information processing structures based on a simplified model of the functioning of the human brain, in which brain cells or neurons, and their interconnections, can perform extremely complex calculations very quickly.

Typical neural networks consist of many processing elements (PEs) interconnected with each other. A neural network has a group of PEs which receive external inputs, and another group of PEs that communicates the output to some external sources. Each PE in the network receives signals from other PEs, weighted by the interconnection weights, and transmits the processed signal to other PEs. Information is thus represented and processed in a parallel and distributed fashion across the network. The major advantages of neural networks are fast processing speed, parallel and distributed representation and processing of information, the ability to be trained to perform non-linear mappings of patterns, and the ability to produce good classification results with imperfect input data.

BACKGROUND: FREEWAY AUTOMATED INCIDENT DETECTION (AID)USING THE PNN

Work on incident detection at UCI has focused on the application of Artificial Neural Networks as a computationally-efficient approach to on-line detection of traffic stream abnormalities. Recent research has demonstrated the potential benefit from and the superiority of approaching the freeway incident detection problem using ANNs with particular emphasis on the Multi-Layer Feed Forward network with back propagation learning (MLF) and our own modified version of a Bayesian Probabilistic Neural Network (PNN). Results have proven the superiority of this approach relative to other existing conventional approaches. The most recent UCI research on freeway incident detection has developed a candidate universal approach that is expected to fulfill a set of universality requirements. This new approach is based on the PNN as a core, and has a preprocessor feature extractor as well as a postprocessor probabilistic interpreter. The overall new framework is a promising neural-network-based UNIversally Transferable Incident Detection (UNITID) algorithm that simultaneously incorporates all the issues of significance to the problem of freeway incident detection including the important transferability issue. The PNN performance was found to be competitive with the MLF in terms of Detection Rate (DR), False Alarm Rate (FAR), and average Time To Detection (TTD). In addition, results also pointed to the possibility of utilizing the real-time learning capability of this new architecture to produce a transferable incident detection algorithm without the need for explicit, developer-attended, off-line retraining in the new site.

The PNN performance was found to quickly improve with time in service, and approach an

"ideal" performance of 100% DR and 0% FAR as shown in Tables 1 and 2, (Abdulhai and Ritchie, 1997). Real time improvement in detection performance is the main contribution of this algorithm. Moreover, the PNN-based framework possesses all the remaining attributes that would make it potentially universal. Ongoing efforts are geared towards the maturation of the algorithms beyond the prototype stage, and preparing them for real-time on-line implementation.

The PNN in the AID system takes as input, a feature vector composed of direct measurements of volumes and occupancies from loop stations at the upstream and downstream ends of the freeway segment over which the AID system operates. On receiving a new input vector, the PNN computes the probability that this input vector relates to an incident condition on the freeway segment. This aposteriori incident probability is computed by first computing likelihood probabilities for incident and normal conditions from the value of the input vector and then using Bayes Theorem. The likelihood probabilities are computed using a Parzen estimation formula. The entire computation can be identified as a 3-layer neural network composed of a pattern layer, a summation layer and an output layer (Abdulhai and Ritchie, 1997).

An extra transformation layer is added before the pattern layer as a modification to the standard PNN. This first layer rotates the input vector so as to transform its coordinates into its principal components. The mathematics of the PNN can be summarized as follows:

Principal components are computed from the input vector and the covariance matrix as follows:

- Let Σ be the covariance matrix associated with the random vector X;
- Let Σ have the eigenvalue - eigenvector pairs:

$$(\lambda_1, e_1), (\lambda_2, e_2), \ldots\ldots\ldots\ldots, (\lambda_p, e_p)$$

where

$\lambda_1 \geq \lambda_2 \geq \ldots\ldots\ldots\ldots\lambda_p \geq 0$;

- The ith principal component of X is given by

$$Y_i = e_i' X = e_{1i} X_1 + e_{2i} X_2 + \ldots\ldots\ldots + e_{pi} X_p$$

and

$$Var(Y_i) = e_i' \Sigma e_i = \lambda_i$$

$$Cov(Y_i, Y_k) = e_i' \Sigma e_k = 0$$

where

i = 1,2,…,p

The weights between the input layer and the transformation layer are the eigenvectors of the sample covariance matrix. The transfer function in the units of the transformation layer simply divides the weighted input to the unit by the standard deviations $\sqrt{\lambda_i}$. The transformed vector Y is then classified as belonging to:

$\pi 1$ if : f1(y) / f2(y) ≥ { [C(1|2) / C(2|1)] * [P2 / P1] }

$\pi 2$ otherwise.

where :

- C(i|j) is the cost of miss-classifying an object as belonging to population πi while it belongs to population πj.
- Pi is the prior probability of occurrence of population πi

Typically, the a priori probability ratio and the cost ratio can be estimated either subjectively or objectively. The Probability Density Functions (PDF) are estimated using Parzen Windows as follows:

$$f_k(y) = \frac{1}{(2\pi)^{p/2}\sigma^p}\frac{1}{m}\sum_{i=1}^{m}\exp\left[-\frac{(Y-Y_{ki})^T(Y-Y_{ki})}{2\sigma^2}\right]$$

where :

k = class or category

i = pattern number

m = total number of training patterns

Yki = ith training pattern from category or population πk

σ = smoothing parameter

p = dimensionality of feature (input) space.

As the algorithm generates an incident alarm, the probability of existence of a true incident in the field is computed using Bayes' theorem as follows:

$$P(I|A) = \frac{P(A|I)\cdot P(I)}{P(A|I)\cdot P(I) + P(A|F)\cdot P(F)}$$

where

A : the event of an incident alarm generation by a given incident detection algorithm.

I : the event of a true incident in the field.

F: the event of an incident-free condition in the field.

P(I): the prior probability of occurrence of an incident.

P(F): the prior probability of an incident-free condition, which is the complement of P(I).

P(A|I): the conditional probability of occurrence (generation) of an alarm given the occurrence of an incident in the field. This could be taken as equal to the correct classification rate of incident related input vectors. It is algorithm-specific, and is obtainable during the testing of the calibrated algorithm.

P(A|F):the conditional probability of occurrence (generation) of an alarm given an incident-free condition in the field. This could be taken as equal to the incorrect classification rate of incident-free input vectors. It is also algorithm-specific, and is obtainable during the testing of the calibrated algorithm.

P(I|A): the conditional probability of existence of a true incident given the occurrence (generation) of an alarm.

Note that P(A|I) and P(A|F) are always less than unity because any incident detection algorithm is not perfectly reliable, and misclassifications inevitably occur.

Similarly, if the incident detection algorithm indicates an incident-free condition, the probability of existence of an incident in the field could be updated using the equation:

$$P(I|\overline{A}) = \frac{P(\overline{A}|I) \cdot P(I)}{P(\overline{A}|I) \cdot P(I) + P(\overline{A}|F) \cdot P(F)}$$

where $\overline{A}$ is the complement of A, i.e. the event of the algorithm is indicating an incident-free condition at a particular time interval. P($\overline{A}$ | I) is also the compliment of the P(A | I) as the conditioning component is the same.

The above scheme is applicable to the output of any incident detection algorithm including the PNN. However, there is another updating process that particularly fits only the PNN. The probability densities f1(X) and f2(X) produced by the PNN after classifying an input vector X can be used in the following equations to update the incident probabilities instead:

$$P(I|X) = \frac{f_1(X).P(I)}{f_1(X).P(I) + f_2(X).P(F)}$$

$$P(F|X) = \frac{f_2(X).P(F)}{f_1(X).P(I) + f_2(X).P(F)}$$

It is noticeable that the above equations are useable right after the probability densities are produced by the PNN and even before a classification of the incoming vector is made. Therefore there is no need to classify each incoming vector but rather update the posterior probabilities directly.

	PNN					
	Before On-Site Patterns Update			After On-Site Patterns Update		
Persistence	DR	FAR	TTD	DR	FAR	TTD
0	31	0.4	1080	100	0.5	15
1	29	0.02	1188	98	0	79
2	29	0	1218	98	0	112
3	27	0	902	98	0	142

Table 1. Performance Improvements of the PNN After Real-Time Updating on the I-880 Patterns.

	PNN					
	Before On-Site Patterns Update			After On-Site Patterns Update		
Persistence	DR	FAR	TTD	DR	FAR	TTD
0	76.12	8.16	415	98.51	0.011	48
1	40.0	2.7	348	98.51	0.009	85
2	29.1	1.3	328	98.51	0.006	116
3	24.6.	0.9	465	97.76	0.003	147

Table 2. Performance Improvements of the PNN After Real-Time Updating of the I35W Patterns.

DESIGN AND IMPLEMENTATION OF THE ONLINE AID SOFTWARE

The AID system is implemented as a Java application. The advantages of implementing incident detection as a Java application are as follows:

1. *Object Orientation*. A Java application is fully object-oriented; the various AID modules are therefore implemented as objects with complete data encapsulation and a meaningful inheritance structure. This makes future program modifications and maintenance an easy task.
2. *Complete Portability*. A Java application program can run on *any* host, which has the Java Virtual Machine (JVM) running. For example, in this project, the AID was developed and tested on a Windows NT platform, but can be executed on any host (say, a Solaris workstation) which has a JVM running; not a single line of code has to be modified to run the AID on a JVM on any platform.

A commonly cited demerit of Java is that Java applications are run by first compiling the source code to an intermediate Java byte-code, which is then interpreted line by line. This makes a Java application run noticeably slower than a C++ application with the same functionality, but which has been compiled to native machine language and executed. However, this slight slowdown in execution is completely inconsequential for AID software since it has to do each round of computation and user-interaction *only once every 30 seconds*. Even with ordinary processor speeds of 100-150 MHz, the Java PNN program can compute each incident probability output and do auxiliary operations such as display, file storage and network input/output all within 2-3 seconds.

Fig. 1 shows the environment in which the AID operates online. Loop data from the Caltrans District 12 are acquired and organized at the D12 headquarters by a CORBA-based Traffic Data Server program as a set of CORBA objects, one for each loop. The CORBA object for each loop holds the speed, volume and occupancy per lane measured by the loop. Fig. 2 gives a module level description of the AID software. In the rest of this section, we explain the functioning of each of these modules. The next section describes the implementation of these modules in the form of Java classes.

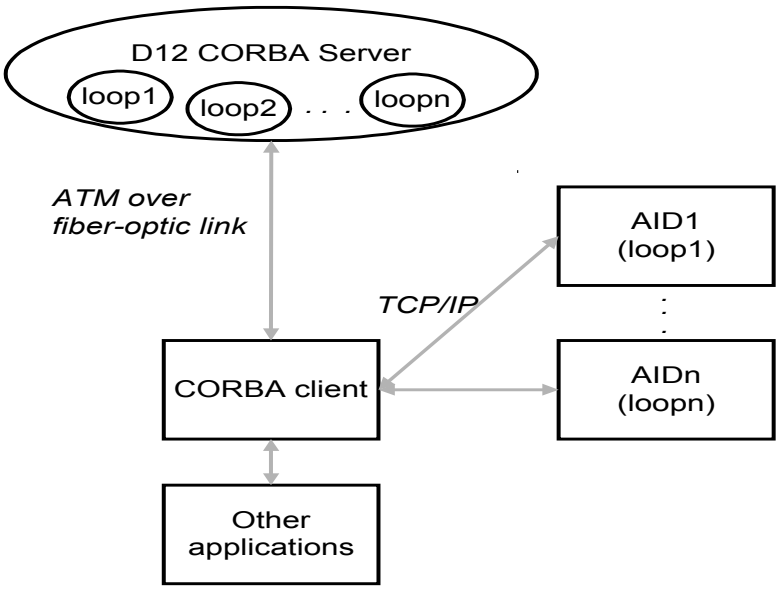


Figure 1. D12 Data Acquisition System

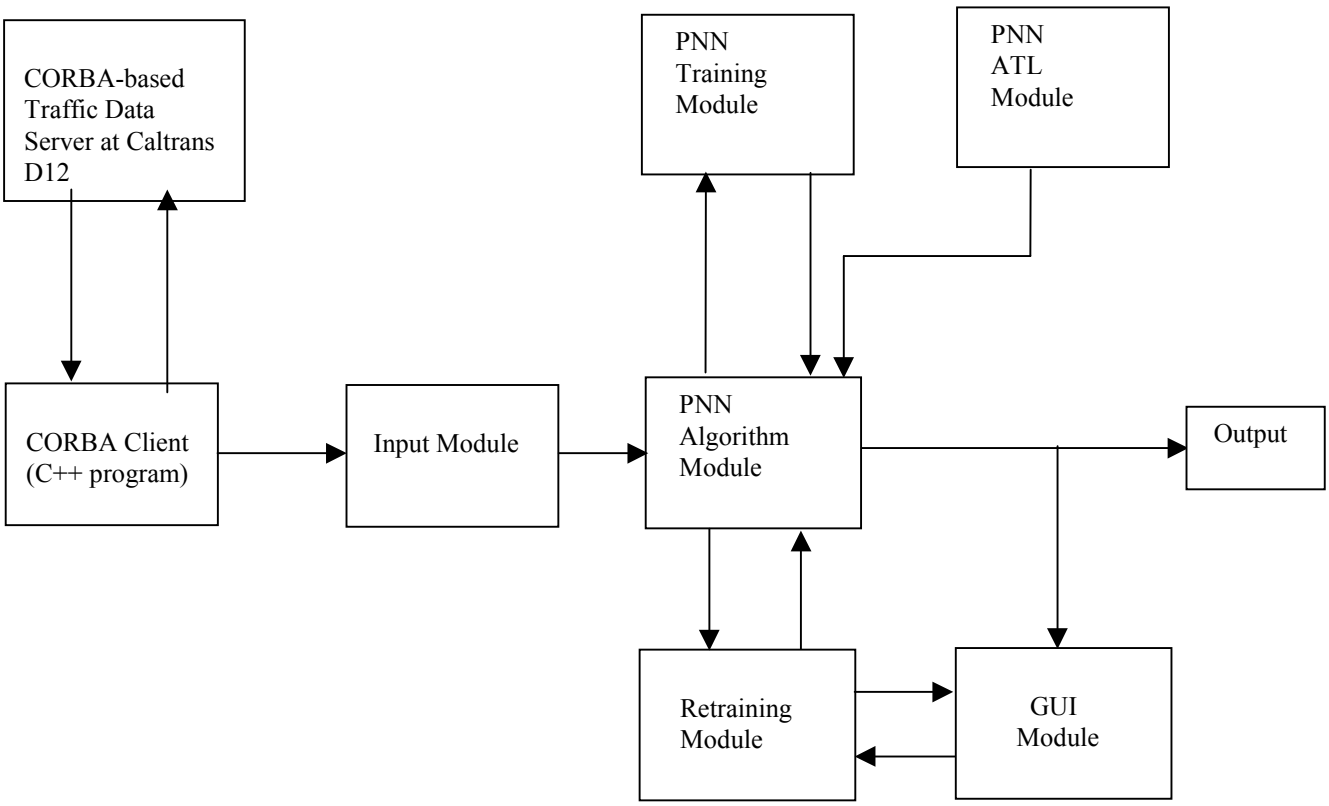


Fig. 2 Modules in the AID software

The CORBA client and AID Input Modules

There are two versions of the input module in the AID software: the file input module and the online-input module. Both modules prepare a 16-dimensional input vector X for input to the PNN algorithm as follows: (the algorithm module actually subtracts the corresponding averages for the current time of the day from these input vector components; it is this *deviation from the average* that forms the actual input to the PNN computation.)

X[0] = Fourth previous upstream occupancy value
X[1] = Third previous upstream occupancy value
X[2] = Second previous upstream occupancy value
X[3] = Previous upstream occupancy value
X[4] = Current upstream occupancy value

X[5] = Fourth previous upstream volume value
X[6] = Third previous upstream volume value
X[7] = Second previous upstream volume value
X[8] = Previous upstream volume value
X[9] = Current upstream volume value

X[10] = Second previous downstream occupancy value
X[11] = Previous downstream occupancy value
X[12] = Current downstream occupancy value

X[13] = Second previous downstream volume value
X[14] = Previous downstream volume value
X[15] = Current downstream volume value

File-input Module

This module can be used to test the PNN with traffic data stored in a file. The module expects

PNN input vectors to be stored in the format described above, serially in the disk file with white spaces or tabs separating the successive input vectors and successive input vector components.

Online-input Module and the CORBA client

The online input module uses a TCP/IP reliable streams connection to a Caltrans real-time traffic data source which is an online loop-data retrieval program running on a separate host. Currently, this real-time traffic source is a C++ CORBA client program that retrieves real-time per-lane traffic data (volume, occupancy and speed) from the CORBA-based Traffic Data Server (TDS), running on a host at the Caltrans D12 headquarters. The CORBA client averages each received traffic data item over all the lanes and sends the averaged data to the AID via a TCP/IP connection. Fig. 1 shows the traffic data flow from the server to the C++ client to the AID system. Every time it receives fresh data from the CORBA client, the online-input module updates the current PNN input vector according to the format described above.

It might be important at this stage to examine how CORBA software is used to make different programs on heterogeneous hosts talk to each other. CORBA software mainly consists of an Object Request Broker (ORB) which acts as a "middleman" between different programs running on different hosts in a network. Also, the CORBA software consists of libraries which can be used to build a *CORBA interface* to any program written in any of the supported languages which includes C++, Java and Smalltalk. Once a program has such an interface, it is called a *CORBA object*. This interface specifies a list of *methods* (procedures) and data variables that other programs (CORBA clients) possibly residing on different hosts in the network can access using the ORB. The CORBA clients do not have to "worry" about what underlying network (TCP/IP, ATM, etc.) protocol is used to connect to the CORBA object or where in the network the object resides or what operating system the object is running on as a program. All these details are taken care of by the ORB. All the client has to do is to know the CORBA interface specifications of the desired CORBA object, use the ORB to first get a *handle* or reference to that object and then subsequently use this handle to invoke the various methods specified in the object interface *as if*

the object was a piece of code compiled and linked to the client program. For example, in the AID software, the entire Traffic Data Server residing on a Caltrans D12 machine really appears to the C++ client program as a set of C++ objects that have been linked to it. Note that a given program can act as both a CORBA client as well as a CORBA object: when it invokes one or more services from other objects, it is acting as a client while it acts a CORBA object when it processes requests that are received from other programs via its CORBA interface.

The following pseudo-code describes how the CORBA client retrieves online data every 30 seconds from the Traffic Data Server.

CORBA client pseudo-code

1. Load the freeway loop list from the **config** file into an array.
2. From the **aid-config** file, load the list of IP addresses and port numbers for the AIDs for each of the desired loops.
3. Get the grand list of CORBA object references (handles) to all freeway loops.
4. Filter the retrieved grand list to obtain the list of handles to the CORBA objects representing only the desired loops (by searching through the grand list for the names of the desired loops.)
5. Every 30 seconds, access the traffic data residing in each of the desired loop objects.
6. For each of the loops, parse the retrieved traffic data string to obtain the per-lane volume, occupancy and speed.
7. Send the average volume, occupancy and speed for each loop to the corresponding AID.

In our implementation, the CORBA client is a program written and compiled in the Microsoft

Visual C++ environment and running on Windows NT. The CORBA software used to connect to the D12 traffic data server is ORBIX, an ORB developed by IONA. The actual network used to connect to the D12 server is a dedicated fiber-optic link with the virtual circuit protocol, ATM, running on top of the physical layer. With CORBA operating on top of such high-speed connections to the real-time data, a distributed ATMS system designer does not have to worry about the networking aspect of the system design and can instead focus solely on the efficient design of the system as a group of CORBA objects and clients. Fig. 2 shows the entire real-time data retrieval set-up including the CORBA modules.

Inside the AID, the Online-input and File-input modules are encapsulated as methods in one single Java class, namely *InputClass.*

PNN Training, ATL and Algorithm Modules

PNN Training Module

The PNN training module reads a training file that stores the complete set of exemplar input vectors that are to be used for training the PNN. Each input vector in the training file has a tag component attached to it that indicates whether that vector corresponds to an incident or a normal condition. The training module reads each of these input vectors from the file, rotates (according to principal component transformation) and then does a min-max transformation of each of the vectors. These final transformed versions of the training input vectors are stored in memory in the form of arrays, *inc* (all incident condition vectors) and *nor* (all normal condition vectors.) These arrays are later used by the algorithm module for computing the likelihood probabilities of an input vector given new data from a freeway segment.

PNN ATL Module

The PNN ATL module is used for computing the volume and occupancy ATLs (Averages for a Time and Location) for the freeway segment on which the AID system is operating. These ATL values are used by the PNN algorithm module in computing volume and occupancy deviations from averages. In the current implementation, the averages are computed for a particular day as

follows: historical volume and occupancy data (at 30-second intervals) for that day are retrieved from the Traffic Data Server using its Web browser interface. These data are averaged over 15 minute intervals, thus giving rise to 24*60/15 = 96 values that are stored in a file (named as the ATL file.)

The ATL module is actually implemented as a separate program that is executed offline. The ATL values it produces in the form of ATL files are read later on by the PNN algorithm module and stored in arrays in the memory. Ideally, the AID user must collect recent historical non-incident data for each day of the week, feed these to the ATL program and thus generate seven ATL files, one for each day of the week. The algorithm module will then automatically choose the correct ATL file during runtime and load the averages for each time of that day.

PNN Algorithm Module

Initially, before starting to receive traffic data inputs, the PNN Algorithm module reads the volume and occupancy averages (ATLs) for its specific freeway segment from the corresponding ATL file. These volume and occupancy ATLs are stored in two respective arrays in the memory.

The algorithm module invokes the input module to retrieve the latest traffic data in the form of the 16-dimensional input vector. In the file-input version of the AID software, the file-input module is invoked, while in the online-input version of the software, the online-input module is invoked.

Given the new input vector, the algorithm module uses the incident and normal PDF functions generated by the training module to compute the conditional (likelihood) probabilities that an incident condition and normal condition will generate this input vector. Using Bayes theorem and the incident and normal likelihood probabilities, the aposteriori probability that the given input corresponds to an incident condition is computed according to the following algorithm (specified in a pseudocode form):

PNN Algorithm Pseudocode

Data*:*

n: Total number of training vectors classified as incident-related

m: Total number of training vectors classified as normal-condition-related

X_k*:* The k^{th} incident-related training vector (all such vectors are stored in an array, *inc)*

Y_k*:* The k^{th} normal-conditon-related training vector (all such vectors are stored in an array, *nor)*

σ: Smoothing parameter in the Parzen estimation formula

p: Dimension of the input vector (p = 16 in our implementation)

X: received input vector

$f_I(X)$: likelihood probability density that an incident condition will generate *X*

$f_N(X)$: likelihood probability density that a normal condition will generate *X*

p_I: apriori probability of an incident condition

p_N: apriori probability of an normal condition; $p_N = 1 - p_I$

P_I : (*PNN Output*) aposteriori probability that given *X*, an incident has occurred on the freeway segment

P(A/I): probability of incident alarm given an incident

P(A/N): probability of incident alarm given a normal condition

MCCR: MisClassification Cost Ratio

Algorithm:

P(A/I)= 0.85;

P(A/N)= 0.04;

// compute likelihood probability for *X* given an incident condition using Parzen's formula

$$f_I(X) = \frac{1}{(2\pi\sigma^2)^{p/2}} \frac{1}{n} \sum_{k=1}^{n} e^{\frac{-\|X - X_k\|^2}{2\sigma^2}} ;$$

// compute likelihood probability for *X* given an normal condition using Parzen's

formula

$$f_N(X) = \frac{1}{(2\pi\sigma^2)^{p/2}} \frac{1}{m} \sum_{k=1}^{m} e^{\frac{-\|X-Y_k\|^2}{2\sigma^2}} \;;$$

if($f_I(X) > MCCR * f_N(X)$) // **incident alarm**

{

// compute probability of incident given *incident alarm:*

$P_I = P_I * P(A/I) / (P_I * P(A/I) + (1- P_I) * P(A/N));$ // **Step 1**

}

else // **normal condition indication**

{

// compute probability of incident given *normal condition indication:*

$P_I = P_I * (1-P(A/I)) / (P_I * (1-P(A/I)) + (1- P_I) * (1-P(A/N)));$ // **Step 2**

}

$P_I = max(min(P_I , 0.95), 0.05);$ // **Step 3**

The output of the PNN is P_I as calculated above. Steps 1 and 2 above are in fact applications of Bayes theorem. In Step 1, P_I is $P(I/A)$, the aposteriori probability that given the incident alarm, an incident has really occurred on the freeway segment. Therefore, by Bayes theorem,

$$P(I/A) = P(I)P(A/I) / P(I)P(A/I) + (1-P(I))P(A/N)$$

where $P(I)$ is the *apriori* probability of an incident on the freeway segment over the observation interval (=30 seconds in our implementation) and $1-P(I) = P(N)$ is the apriori probability of a normal condition on the freeway segment. In Step 1, the previous output, i.e. the aposteriori probability P_I computed for the *previous* input is itself used as the apriori probability, $P(I)$ in the above Bayes formula. Thus, Bayes theorem is used recursively. The initial value of $P(I)$ (used for computing the very first output) can be specified by the user when the AID software starts

execution; the user is prompted to supply the value of *P(I)*.

The event, *A*, an incident alarm, is generated if $f_I(X) > f_N(X)$. This is nothing but the maximum likelihood (ML) criterion for deciding that an incident has occurred. If we replace this with a different criterion for generating an incident alarm, then we will obtain a correspondingly different algorithm for incident detection; steps 1 and 2 will however remain unchanged. What steps 1 and 2 do is to convert a *hard decision* on the occurrence of an incident (alarm/no-alarm) into a *soft decision*, i.e. probability of occurrence of an incident. In the Bayes formula, the probabilities *P(A/I)* and *P(A/N)* actually depend on the accuracy of the specific criterion used for generating an incident alarm. In our case, the fixed values of *P(A/I)=0.85* and *P(A/N)=0.04* have been found to satisfactorily reflect the accuracy of ML criterion. The actual algorithm above shows a factor, *MCCR,* the MisClassification Cost Ratio used in the ML criterion. *MCCR* is used to give different weights to minimizing the probability of a false alarm versus minimizing the probability of a miss (deciding a no-incident when there is an incident) since there is a tradeoff between minimizing these two probabilities. *MCCR* is initially set to 1 when the AID starts, but can later be changed according to the value specified by the user; a slider button is moved on the display screen by the user to set *MCCR* to the desired value.

A similar explanation holds for the computation in Step 2, where P_I is really *P(I/NC)*, the aposteriori probability that given the normal condition indication (*NC*), an incident has really occurred on the freeway segment.

Step 3 in the algorithm bounds the output probability between 0.05 and 0.95 to ensure that if the output calculated in step (1) reaches 0.0 or 1.0, it does not get locked at that value.

The algorithm module first does a principal component transformation of the received input vector and before feeding it to the above algorithm. The exemplar (training) vectors, $\{X_k\}$ and $\{Y_k\}$ have also been similarly transformed after being retrieved from the training file, as explained above in the description of the training module. The motivation behind doing this transformation before employing Bayes theorem is explained in Abdulhai and Ritchie (1998).

The PNN algorithm module also performs additional operations, namely invoking the GUI module to display the latest input and output.

The PNN training and algorithm modules are encapsulated as separate methods (*train() and compute_output()* respectively*)* in one single Java class, namely *pnnThread.* This is a Java Thread class. A single object of this class is instantiated when the AID system starts functioning. That is, a *pnnThread* thread is created and started by the main method of the *IDet* class which is the primary class of the AID system. The *pnnThread* thread runs concurrently with the other thread created by the Idet class, namely, the display thread, *idet_applet.* The *run()* method of the *pnnThread* object carries out the initial PNN training by calling the *train()* method. After that, the *run()* method loads the ATL values from the corresponding files and then enters an infinite loop. Each cycle of this loop consists of:

1. acquiring the latest input from the input module
2. acquiring the latest user-specified MisClassification Cost Ratio (MCCR) from the GUI module,
3. invoking the *compute_output()* method which is the Java implementation of the PNN algorithm module.

PNN Retraining Module

The retraining module is invoked by the algorithm module whenever the latter classifies an output as a suspected incident according to the Incident Threshold value (certain cut off probability, specified by the user, above which the field event is considered a certain incident). The retraining module then prompts the user with the computed incident probability value and asks the user to verify whether this is an incident, using cameras or police records or similar. If the user classifies this as an incident condition, the current input vector is appended to a temporary incident data file. The user also indicates whether retraining is to be done; if so, the user also specifies the time interval whose incident data is to be incorporated into the training file. The retraining module accordingly retrieves the specified incident data from the temporary file and this data either *replaces existing data* in the training file *or is appended* to the training

file according to what the user specifies. The above described interaction with the user is done using a GUI as shown in fig. 4. In the current implementation, the Incident Threshold value that triggers the retraining module has been hard coded to 0.5 in the program. However, in a future version, the user will be able to set this value (among other things) in a config file. When the AID starts, it will configure itself according to the specifications in the config file.

The retrain module is implemented as a thread, *retrain_thread*, that runs in parallel with *pnn_thread*. Thus, the retraining of the PNN, i.e., computation of the new *inc* and *nor* arrays continues in the background *without disturbing the operation of* ***pnn_thread***. As soon as *retrain_thread* has finished computation of the new *inc* and *nor* arrays, it overwrites the corresponding old arrays in the data space of *pnn_thread.*

GUI Module

The GUI for the AID system consists of a display of the probabilistic neural network (PNN), a sliding window display of the volume and occupancy deviation inputs to the PNN algorithm module and a sliding window display of the output (incident probability). Figures 3a, 3b, and 3c show screenshots of the GUI for the AID running on a Windows NT host, for typical normal, incident beginning and incident end conditions respectively. The two windows on the left show running plots of real-time volume and occupancy deviations from ATL values. Plots for upstream variables are drawn in red, while those for downstream variables are drawn in blue. The window on the right shows the running plot for the incident probabilities computed every 30 seconds. The "Output Only" button on the top right corner of the GUI can be used to toggle between the elaborate display shown and a reduced display showing only the plot of the incident probability.

The GUI module is implemented as an object of the Java Applet class. The GUI display is updated whenever the PNN algorithm module calls the *repaint()* method of this object. Each sliding window plot (upstream & downstream volume deviations/ upstream & downstream occupancy deviations/ output probability) is maintained as a separate object of a class called the *OutputArrayClass.* This class encapsulates the methods for maintaining the running window plot data as a circular queue.

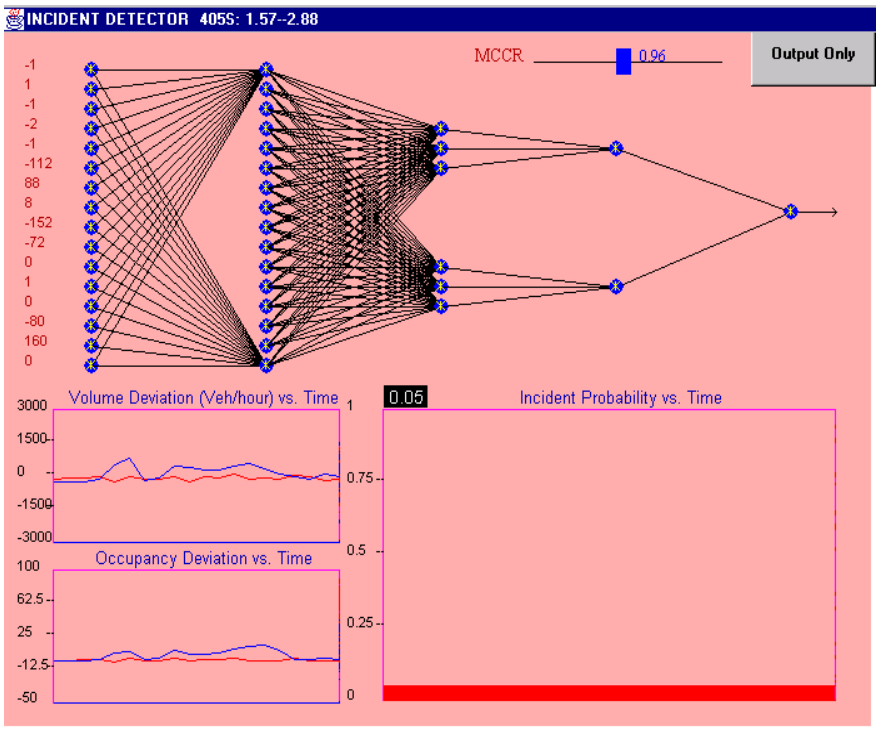


Fig. 3a. Snapshot of AID screen during normal conditions

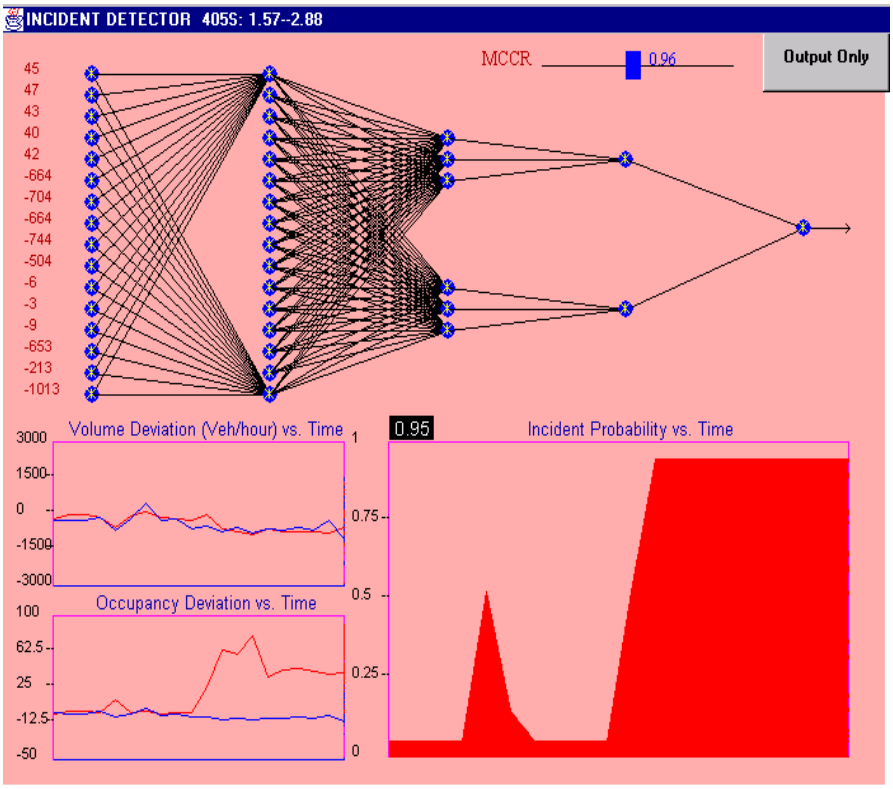


Fig. 3b. Snapshot of AID screen during beginning of incident

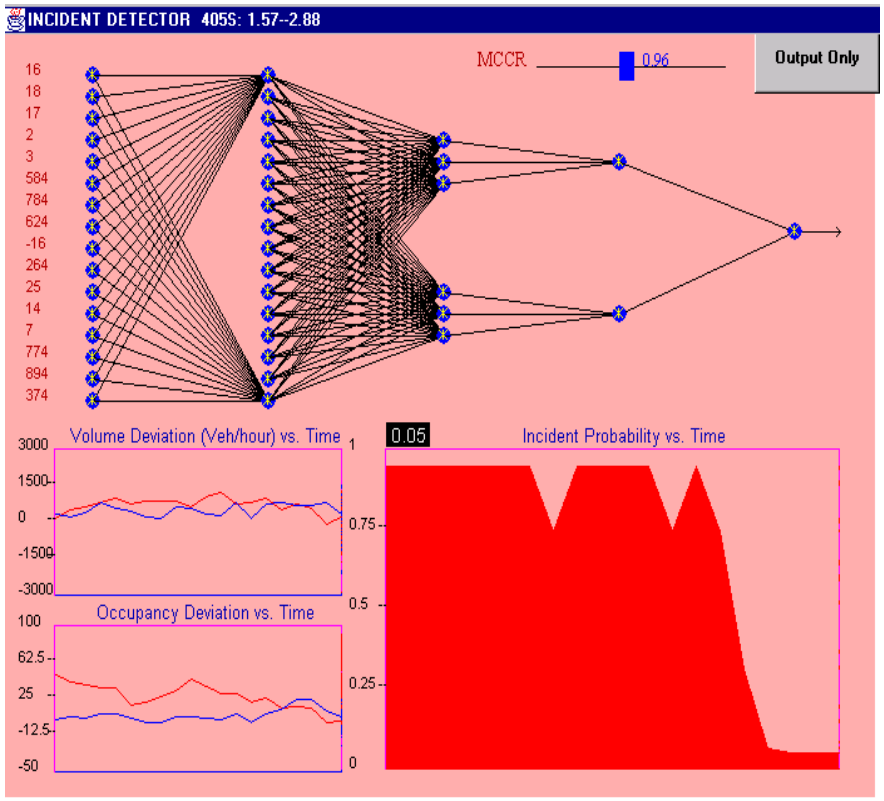


Fig. 3c. Snapshot of AID screen towards end of incident

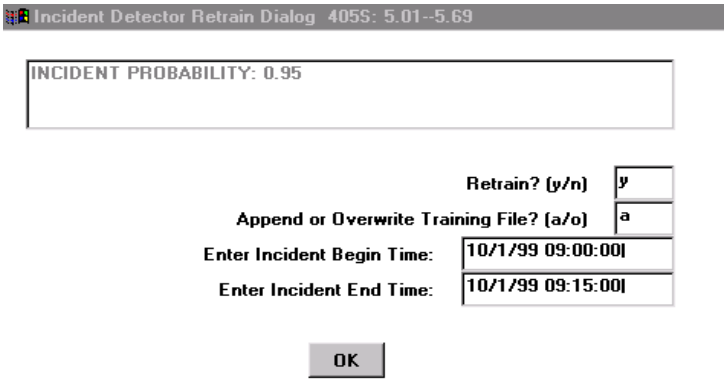


Fig. 4. Snapshot of GUI for PNN Retraining

DETAILED DESCRIPTION AND PSEUDOCODE OF THE AID SOFTWARE

This section describes the organization of the AID software in the form of Java classes. The AID classes are listed below followed by the pseudocode for the important classes, viz. IDet, pnnThread, RetrainDialog and RetrainThread.

1. IDet: This is the "main" class (the class that contains the main() function). In Java, the compiled program (the executable) has the same name as its main class.
2. IncidentDetector: This is the GUI class of IDet and is an Applet class. It contains methods for updating the display of IDet (the PNN, the input and output plots) and for accepting input from the user.
3. pnnThread: This is the core class of IDet. It is a Java Thread class. The main *get_next_input-compute_output* endless loop of the software is contained in its run() method.
4. inputClass: contains methods for connecting to the source of input data (either a local file or a TCP/IP socket stream connection to a remote host.)
5. OutputClass: This class contains methods for maintaining and displaying a sliding window plot of variables that evolve with time. These methods are called by the IncidentDetector class in order to display a sliding window plot of the output versus time as well as upstream and downstream volumes and occupancies versus time.
6. statClass: This is the 'statistics' class that contains methods for computing the mean, standard deviation and the eigenvectors of the correlation matrix of the exemplar input vectors used for training the PNN. These methods are called by the training module of Idet , i.e. the *train()* method of the pnnThread class.
7. RetrainDialog: This class implements the GUI for retraining of the PNN. It contains methods for pre-processing the different user inputs regarding retraining and then sending these user data to the RetrainThread object which in turn carries out the retraining.
8. RetrainThread: This is a Thread class and carries out the retraining of the PNN by basically invoking the *train()* method of the PNN after the new training data have been incorporated into the training file.

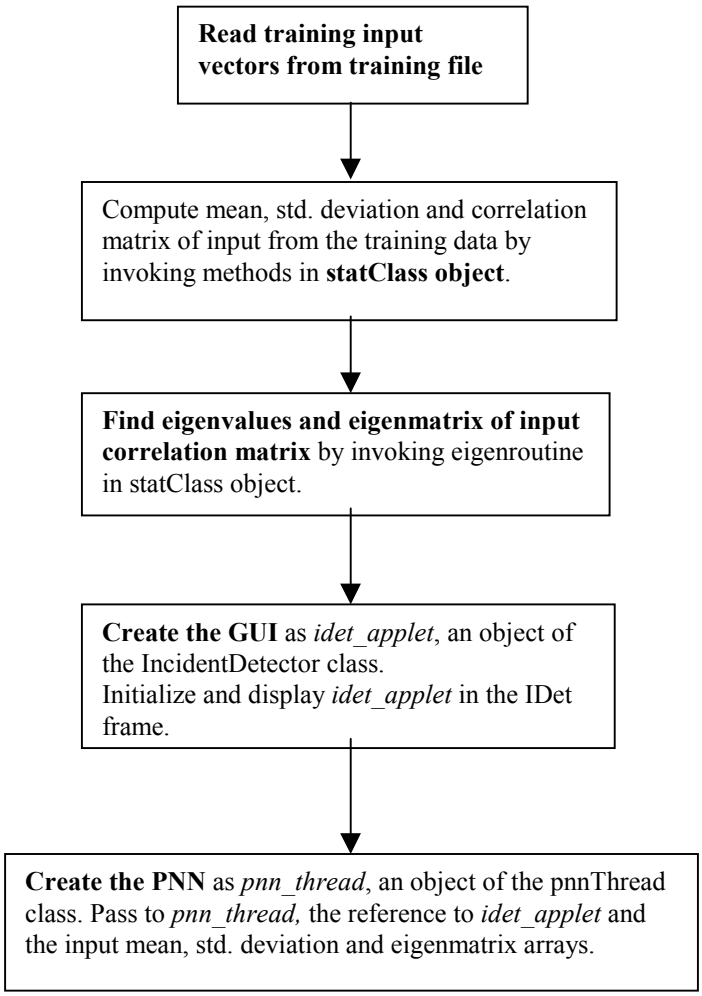
In the rest of this chapter, we describe the pseudo-code for the important methods of the

important AID classes in detail in the form of program control-flow-charts.

Class Idet

This class is the 'main' class of the program, i.e. it contains the *main()* method. Program execution starts from the main() method. IDet is a Java Frame class. The PNN is created as a thread, *pnn_thread(),* by *Idet.main()*. Also, the display (GUI) is created by *Idet.main()* as an Applet object of the IncidentDetector class.
Important methods in IDet are: *main()*.

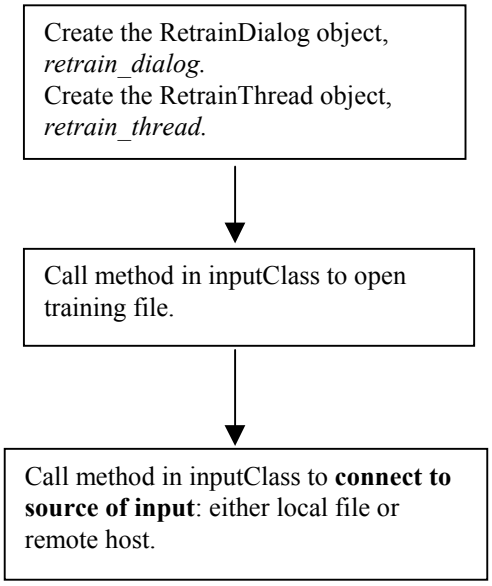
Pseudocode for *main()*:



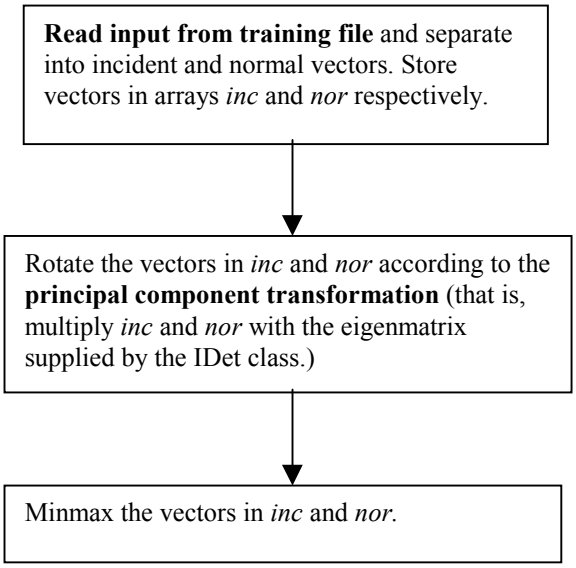
Class pnnThread

Main methods are: the constructor, *pnnThread(), run(), train()*, *compute_output()* and *update_training_file()*.

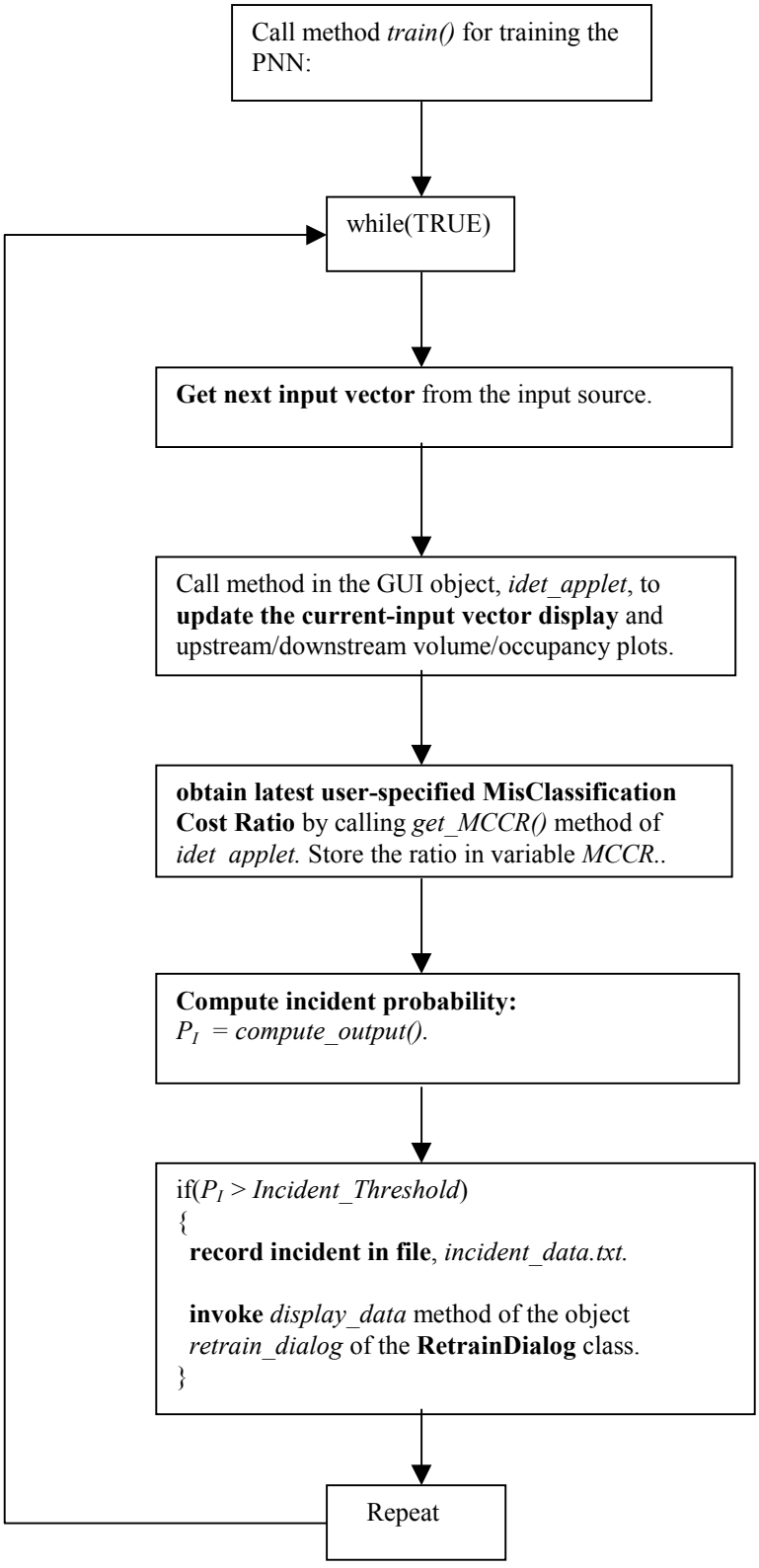
Pseudocode for constructor, *pnnThread():*



Pseudocode for *train():*

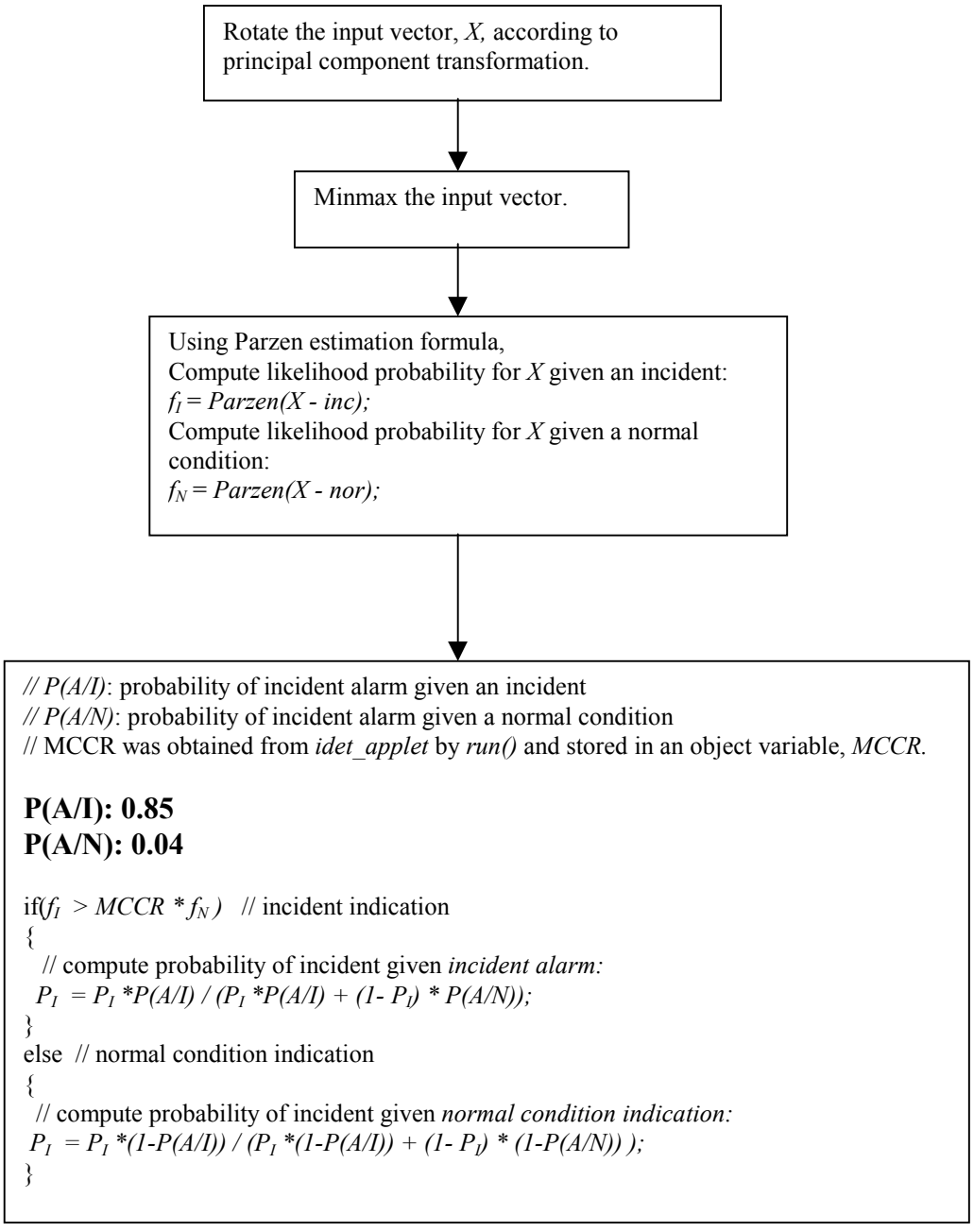


Pseudocode for *run()*:



Pseudo-code for *compute_output():*

(refer to Appendix A to gain a full understanding of the algorithm for computing the incident probability.)



Pseudocode for *update_training_file(boolean append, time data_interval):*

(This method is called by *retrain_thread.)*

```
Retrieve all the data from file, incident_data.txt, that had been
recorded within the time interval, data_interval.
if (append is true)
  Append the new data to the end of  incident_data.txt
else
  Overwrite the entries at the beginning of incident_data.txt
  with the new data.
```

Class RetrainDialog

This class is derived from the Java Frame class. It creates a display window on the screen for interaction with the user regarding retraining the PNN.

Main methods are: *RetrainDialog()* (the constructor) and *action().*

Pseudocode for constructor, *RetrainDialog():*

```
Create text field for displaying new incident data
Create text field for accepting the Retrain/No-Retrain input from
user.
Create text field for accepting the Append/Overwrite input from the
user.
Create text fields for accepting the user input: the time interval
whose incident data are to be used for retraining.
```

Pseudocode for *action(Object arg):*

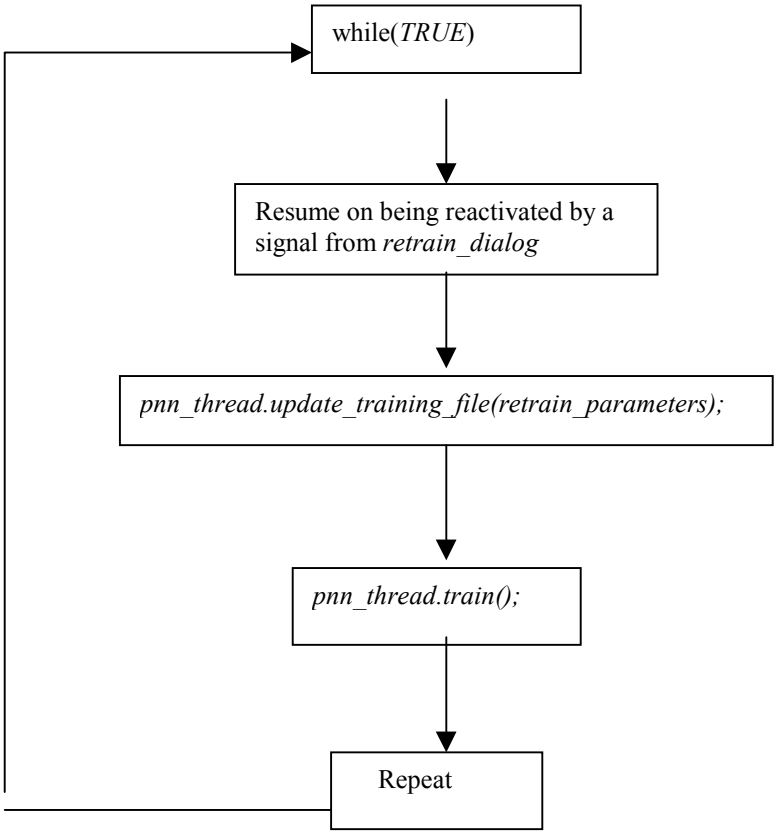
```
if(arg == "OK")  // user has finished specifiying the retrain parameters
{
 if(retrain_comand == "YES")  // user has requested a retraining
 {
  Pass all the following retraining parameters to retrain_thread:
    Append/Overwrite flag
   Begin Time for the new incident training data
   End Time for the new incident training data

  Reactivate retrain_thread.
 }
}
else   // user has specified "QUIT"
  close the retrain_dialog window.
```

Class RetrainThread

This class is derived from the Java Thread class. A single thread object of this class, *retrain_thread,* is created by the constructor of the pnnThread class (refer to the corresponding pseudocode given above.) The thread, *retrain_thread,* runs in parallel with *pnn_thread*. Thus, the retraining of the PNN, i.e. computation of the new *inc* and *nor* arrays continues in the background **without disturbing the operation of *pnn_thread***. As soon as *retrain_thread* has finished computation of the new *inc* and *nor* arrays, it overwrites the corresponding old arrays in the data space of *pnn_thread.*

Main methods are: *run().*



SUMMARY AND CONCLUSIONS

In this project, we extended existing freeway incident detection research conducted under both PATH and under the ATMS Testbed Research Program, to operationaliz its principal findings. The most promising neural network to date, the PNN, was integrated into the UCI testbed for on line operation on the testbed network in Southern California.

The PNN incident detection system was re-coded in Java, to facilitate network communications and platform-independent operation. A Java-based graphical user interface has been developed. The GUI components include a display of the probabilistic neural network (PNN), the current input to the PNN, a sliding window display of the output (the computed incident probability every time step) and a sliding button to allow the user to specify the desired misclassification cost ratio. The GUI code is in the form of a Java Applet object and has a modular structure that makes it easier to incorporate possible future modifications and extensions. The PNN algorithm itself was then translated from C to Java as a stand alone application object and was interfaced to the GUI applet running on the same host. The GUI display is updated each time a new output is computed by the PNN. The PNN algorithm and the GUI display update run as separate threads of control in Java; this concurrency leads to better utilization of CPU resources. A new module for computing the principal component transformation of the volume and occupancy inputs was developed to replace using statistical packages for this transformation. This was needed for maximum portability and independence of the overall system. Another module for computing volume and occupancy historical Averages for different Times and Locations (ATLs.) was also developed to prepare the ATLs from real freeway data. The PNN and GUI were tested and correct operation was confirmed with sample inputs from data files.

The whole package was interfaced to a remote C++ CORBA client program that acquires online CalTrans traffic data from a CORBA server in the Testbed. Communication modules were added to the CORBA client program as well as the PNN to enable online volume and occupancy data from different freeway sections to be sent from the CORBA client to the PNN at a specific rate

(once every 30s). The data are sent to the PNN using a reliable TCP/IP streams sockets connection.

An on-line retraining module was developed as well. This module enables the TMC operator to initiate retraining on recently captured incident data, on-line without disturbing the operation of the system.

The PNN was then started on-line on a 5 mile section of the 405 freeway, for on line monitoring and testing. The overall on-line operation of the PNN was demonstrated to Caltrans engineers from D12. Currently, efforts are in progress to expand the network coverage to enhance the odds of capturing incidents. On line evaluation will be performed using the developed on-line PNN, under the University of California Irvine ATMS Testbed.

APPENDIX A: BRIEF INTRODUCTION TO CORBA CONCEPTS

CORBA is an acronym for **C**ommon **O**bject **R**equest **B**roker **A**rchitecture. It provides a high-level framework for developing object-oriented distributed applications. Developing distributed applications becomes significantly easier within such a framework. In fact, distributed applications interact as if they were all implemented within a single computer in a single programming language.

CORBA defines a standardized architecture for Object Request Brokers (ORBs.) An ORB is a software component that mediates the transfer of messages from a *client* program to a *CORBA* object. Fig. 5 illustrates how an ORB hides the complexity of the underlying network communications from the programmer [4]. When a client invokes a member function on a CORBA object, the ORB intercepts the function call and redirects it to the target object. The latter returns the results (output) of the function call to the ORB which in turn relays the results to the client. Part of the ORB resides on the same host as the CORBA object and the other part resides on the client host. Thus, the ORB is really a collection of software modules residing on all hosts. Every CORBA object has a standard interface specified in the Interface Definition Language (IDL). The IDL interface of a CORBA object consists of a list of the data variables (or *attributes)* and member functions (or *operations*) of the object; client programs can make calls to these member functions or access the data variables by simply sending an appropriate command to the ORB without having to know the specific language in which the object is implemented or the host operating system on which the object resides. A simple example is given below to explain the various steps involved in creating and accessing CORBA objects.

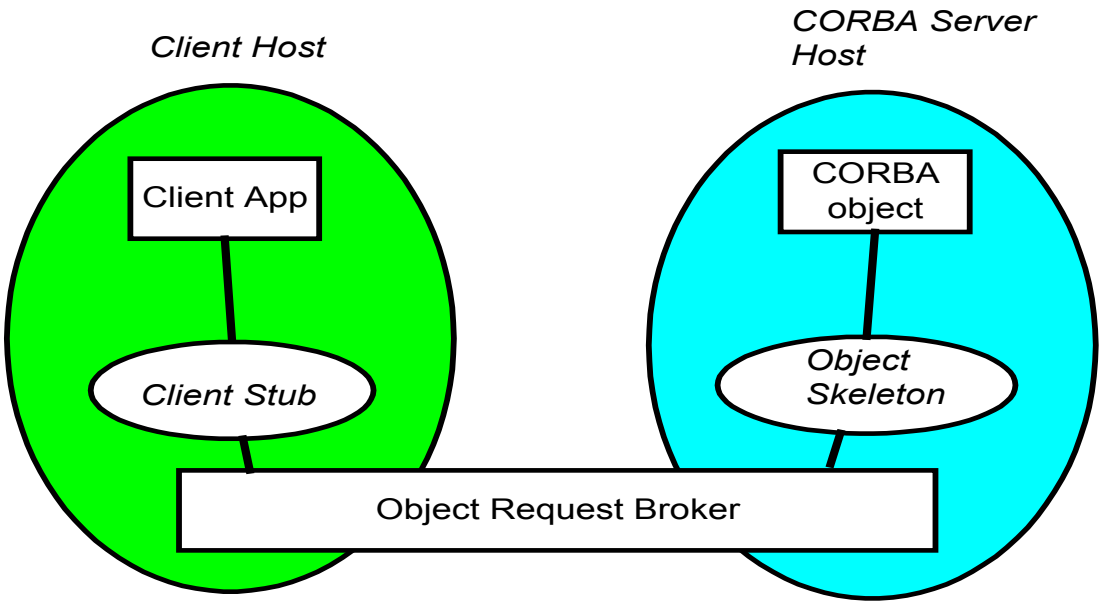


Fig. 5. The CORBA architecture

Example: Encapsulating freeway loop data as a CORBA object using the ORBIX ORB:

To further illustrate how CORBA objects can be developed and accessed by clients, we consider developing a CORBA object holding data from a freeway loop in the C++ language using the CORBA software called ORBIX from Iona Corp. [4]. In this simple example, the freeway loop object must hold 3 variables, volume, speed and occupancy.

1. *IDL file*

The first step is to define a public interface to the loop object in the IDL language in a file with a ".idl" extension (lets call it loop.idl) as follows:

// File loop.idl

```
interface Loop
{
        readonly attribute long volume;
        readonly attribute float speed;
        readonly attribute float occupancy;
}
```

The keyword readonly means that the clients can only read but not set the attribute values.

2. *Client-stub* and *Object-Skeleton-Code*

The above IDL code is then compiled using the ORBIX IDL compiler for C++, thus generating C++ code consisting of a header file, loop.hh (2 versions, one for client side and the other for server side) and two C++ source files, loopC.c and loopS.c. The file loop.hh for the client side contains the Loop class that defines the client's view of a CORBA loop-data object. The class Loop_var is a helper class for Loop that helps manage memory for objects of type Loop in a client program. The files loop.hh and loopC.c form the *client-stub*. A client program that accesses objects of type Loop must be compiled and linked with loopC.c and must include loop.hh.

On the server side (as shown in fig. 5), the files loop.hh and loopS.c form the *object-skeleton-code*.

//File loop.hh for Client Stub

```
#include <CORBA.h>

class Loop: public virtual CORBA::Object
{
        public:
                virtual CORBA::Long volume(CORBA::Environment
                 &IT_env=CORBA::IT_chooseDefaultEnv())
                 throw (CORBA::SystemException);
```

```
        public:
                virtual CORBA::Float speed(CORBA::Environment
                 &IT_env=CORBA::IT_chooseDefaultEnv())
                 throw (CORBA::SystemException);
        public:
                virtual CORBA::Float occupancy(CORBA::Environment
                 &IT_env=CORBA::IT_chooseDefaultEnv())
                 throw (CORBA::SystemException);

}
class Loop_var: public CORBA::_var
{
        public:
                Loop_var &operator= (Loop *IT_p);
                Loop_var &operator= (const Loop_var &IT_s);
                Loop* operator-> ();
}
```

//File loop.hh for Object-Skeleton-Code

```
#include <CORBA.h>

class LoopBOAImpl: public virtual Loop
{
        public:
                virtual CORBA::Long volume(CORBA::Environment
                 &IT_env=CORBA::IT_chooseDefaultEnv())
                 throw (CORBA::SystemException) = 0;

                virtual CORBA::Float speed(CORBA::Environment
```

```
                &IT_env=CORBA::IT_chooseDefaultEnv())
                throw (CORBA::SystemException) = 0;

               virtual CORBA::Float occupancy(CORBA::Environment
                &IT_env=CORBA::IT_chooseDefaultEnv())
                throw (CORBA::SystemException) = 0;
}
```

The class LoopBOAImpl in the server-side object-skeleton code defines the abstract member functions that the loop-data CORBA object class must implement in order to implement the IDL interface Loop.

3. *Loop-data class implementation*

To implement the IDL interface Loop, we must define a C++ class that inherits from abstract class LoopBOAImpl and implements the abstract member functions of LoopBOAImpl. Therefore, we define the implementation class in a header file, loop_i.h as follows:

```
// File loop_i.h
#include <Loop.hh>
class Loop_i: public virtual LoopBOAImpl
{
        CORBA::Long m_volume;
        CORBA::Float m_speed;
        CORBA::Float m_occupancy;

public:
        Loop_i();
        virtual ~Loop_i();

        // functions corresponding to the IDL attributes
```

```
        virtual CORBA::Long volume(CORBA::Environment&);
        virtual CORBA::Float speed(CORBA::Environment&);
        virtual CORBA::Float occupancy(CORBA::Environment&);

        void set(CORBA::Long vol, CORBA:Float sp, CORBA:Float occ);
}
```

The functions defined in class Loop_i are then implemented in the file loop_i.c as follows:

```
// File loop_i.c
#include "loop_i.h"

CORBA::Long Loop_i::volume(CORBA::Environment&)
{
        return(m_volume);
}

CORBA::Float Loop_i::speed(CORBA::Environment&)
{
        return(m_speed);
}

CORBA::Float Loop_i::occupancy(CORBA::Environment&)
{
        return(m_occupancy);
}

void set(CORBA::Long vol, CORBA:Float sp, CORBA:Float occ);
{
        m_volume = vol;
```

```
        m_speed = sp;
        m_occupancy = occ;
}
```

4. *Creating a loop-data CORBA server object of class* Loop_i :

Creating a CORBA object of class Loop_i inside a function (say main()) is simply done by constructing a C++ object of type Loop_i . Once created, the object is available to clients. However, to process IDL calls from clients, the Orbix function, CORBA::BOA::impl_is_ready() must also be called:

```
#include "loop_i.h"
int main()
{
        Loop_i loop1();  // CORBA object created
        try
        {
                CORBA::Orbix.impl_is_ready("LoopSrv", 100000L);
        }
        catch (CORBA::SystemException &se)
        {
                cerr << "impl_is_ready() failed" << endl;
                cerr << "Exception:" << endl << &se;
                return 0;
        }
        .
        .
        .
}
```

The first parameter to the function impl_is_ready() specifies the CORBA name of the object. This is the name that is used by a client in order to obtain a handle to the object .The second parameter specifies a timeout period (in milliseconds) for which the impl_is_ready() call should block while waiting for an IDL operation call from the client. If a client call arrives during this period, Orbix calls the appropriate member function of the implementation object and resets the timer to 0. But if no client call arrives during this period, impl_is_ready() simply returns.

5. *A CORBA client that accesses a* Loop_i *object:*

To obtain a handle (or reference) to a Loop_i object named "LoopSrv", a client program can "bind" to that object as shown in the code below. The bind call really creates a local proxy object that acts as a double for the remote implementation object.

```
// CORBA client
#include "loop.hh"
#include <iostream.h>

int main()
{
        Loop_var loop1;

        try
        {
                loop1 = Loop::_bind(":LoopSrv");
        }
        catch(CORBA::SystemException &se)
        {
                cerr << "bind failed" << endl;
                cerr << "Exception:" << endl << &se;
                return 0;
```

```
        }
        try
        {
                cout << "Volume = " << loop1->volume << endl;
                cout << "Occupancy = " << loop1->occupancy << endl;
        }
        catch(CORBA::SystemException &se)
        {
                cerr << "failed: cannot access loop1 attributes" << endl;
                cerr << "Exception:" << endl << &se;
                return 0;
        }
        .
        .
        .
}
```

At the end of these five steps, we have developed a server application that creates a loop-data CORBA object and also developed a client application that can access the data in a loop-data object without having to worry about what host in the network the server application is residing on or what operating system it is running on. The actual compilation and creation of the client and server executables needs linking to the appropriate Orbix libraries; for details on these steps, we refer the reader to the Orbix Programmers Guide [4].

REFERENCES

[1] Abdulhai B., and Ritchie S.G. (1997), "Development of a Universally Transferable Freeway Incident Detection Framework", Presented at the TRB 76th Annual Meeting, preprint #97112.

[2] Abdulhai B., "A Neuro-Genetic-Based Universally Transferable Freeway Incident Detection Framework", PhD Dissertation, University of California, Irvine, 1996.

[3] Johnson, R.A., and Wichern, D.W., "Applied Multivariate Statistical Analysis", Prentice Hall, Inc., 1992.

[4] "Orbix Programmer's Guide", Iona Technologies PLC, Oct . 1997.