Implementation of Distributed Key Generation Algorithms using Secure Sockets *

A. T. Chronopoulos[†] F. Balbi, D. Veljkovic, N. Kolani
Dept. of Computer Science,
University of Texas at San Antonio,
6900 North Loop 1604 West, San Antonio, TX 78249,
atc@cs.utsa.edu

Abstract

Distributed Key Generation (DKG) protocols are indispensable in the design of any cryptosystem used in communication networks. DKG is needed to generate public/private keys for signatures or more generally for encrypting/decrypting messages. One such DKG (due to Pedersen) has recently been generalized to a provably secure protocol by Gennaro et al. We propose and implement an efficient algorithm to compute the (group generator) parameter g required in the DKG protocol. We also implement the DKG due to Gennaro et al. on a network of computers using secure sockets. We run tests which show the efficiency of our implementation.

1 Introduction

1.1 Preliminaries

In order to achieve secure communications between several servers (called players) running on (different) computers by exchanging messages (which are encrypted/decrypted) public/secret keys are required. DKG algorithms aim to generate and distribute the keys in a secure fashion. A corrupted player is one who is faulty for some reason (e.g. system malfunction or external attack) and thus the messages which he sends should not be trusted as correct by the other players. If the number of corrupted players exceeds a certain threshold then the DKG algorithm fails to generate and distribute the keys for secure communication.

DKG protocols are used in the design of discrete logarithm based cryptosystems for secure communication in computer networks (see [1],[2],[3],[4],[5],[6] and references therein).

The simplified architectural model consists of n servers (called $players$) which cooperate to generate two keys, one public and one private. The public-key is made public and the private-key is kept secret and it is shared by the players. The shared private-key is guaranteed to be secret assuming that an adversary player can only compromise fewer than n/2 players. This secret key is used to encrypt/decrypt messages or compute signatures.

In Shamir's discrete-log based secret sharing protocol a trusted party is used to generate a random value x (the secret key) and (the public key) y = g^x is made public. In the DKG each (of n) players shares a randomly chosen secret x_i. The secret-key x is the sum of the shared x_i's. The public-key $y = g^x$ is the product of the $y_i = g^{x_i}$.

We make the following assumptions for the distributed system model implementing the DKG protocol.

Players : We assume n players running on different computers connected via a secure point-to-point network. We also assume that the network has a secure broadcast capability.

Network communication: We assume a partially

*This research was supported, in part, by a grant from the Center for Infrastructure Assurance and Security at The University of Texas at San Antonio and by NSF CCR-0312323.

†Senior member IEEE, Corresponding author

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on May 13,2010 at 19:21:39 UTC from IEEE Xplore. Restrictions apply.

synchronous communication model in which the messages are delivered within a maximum fixed time bound.

Faulty Player : A player which is corrupted can act as an adversary. He can send and receive messages to/from other players. In the worst case an adversary could communicate last in each communication round (called rushing adversary).

Modular Arithmetic: We assume that all the players obtain an initial input from a trusted party of a triple (p, q, g), where p and q are primes (q is the greatest divisor of $p-1$) and g a generator of the subgroup G_q of order q in Z_p^*. The protocol computes a private-key x in Z_q and a public-key $y = g^x modp$.

Feldman ([2]) introduced the verifiable secret sharing protocol (called Feldman VSS). It uses a trusted dealer (who is assumed never to be corrupted) to share a (secret) key x among n parties. Pedersen ([5]) designed a distributed protocol (Ped-DKG) to achieve the distribution of the key x without a central dealer. It consists of n parallel runs of Feldman-VSS. Each player selects a random secret (share) x_i and executes Feldman-VSS to share it with others. The secret x (which need not be computed) is the sum of the (uncorrupted) players secret shares. Gennaro et al ([4]) showed that the Ped-DKG protocol may be insecure under some attacks. They proposed a secure protocol (called here GJKR-DKG protocol from the initials of the last names of the authors).

We next present the Ped-DKG and GJKR-DKG protocols.

1.2 Distributed Key Generation Protocol: Ped-DKG

1. Each player P_i chooses a random polynomial $f_i(z)$ over Z_q of degree t:

$$f_i(z) = a_{i0} + a_{i1}z + \ldots + a_{it}z^t$$

   P_i broadcasts $A_{ik} = g^{a_{ik}} \bmod p$ for $k = 0, \ldots, t$. Denote a_{i0} by z_i and A_{i0} by y_i. Each P_i computes the shares $s_{ij} = f_i(j) \bmod q$ for $j = 1, \ldots, n$ and sends s_{ij} secretly to player P_j.

2. Each P_j verifies the shares he received from the other players by checking for $i = 1, \ldots, n$:

$$g^{s_{ij}} = \prod_{k=0}^{t} (A_{ik})^{j^k} modp \tag{1}$$

   If the check fails for an index i, P_j broadcasts a *complaint* against P_i.

3. If more than t players complain against a player P_i, then that player is clearly faulty and therefore disqualified. Otherwise P_i reveals the share s_{ij} matching Eq. 1 for each complaining player P_j. If any of the revealed shares fails this equation, P_i is disqualified. We define the set $QUAL$ to be the set of non-disqualified players.

4. The public value y is computed as $y = \prod_{i \in QUAL} y_i \bmod p$. The public verification values are computed as $A_k = \prod_{i \in QUAL} A_{ik} \bmod p$ for $k = 1, \ldots, t$. Each player P_j sets his share of the secret as $x_j = \sum_{i \in QUAL} s_{ij} \bmod q$. The secret shared value x itself is not computed by any party, but it is equal to $x = \sum_{i \in QUAL} z_i \bmod q$.

1.3 Distributed Key Generation Protocol : GJKR-DKG

Generating x:

1. Each player P_i performs a Pedersen-VSS of a random value z_i as a dealer:

   (a) P_i chooses two random polynomials $f_i(z)$, $f_i^{'}(z)$ over Z_q of degree t:
   $f_i(z) = a_{i0} + a_{i1}z + \ldots + a_{it}z^t$
   $f_i^{'}(z) = b_{i0} + b_{i1}z + \ldots + b_{it}z^t$
   Let $z_i = a_{i0} = f_i(0)$. P_i broadcasts $C_{ik} = g^{a_{ik}} \hbar^{b_{ik}} \bmod p$ for $k = 0, \ldots, t$. P_i computes the shares $s_{ij} = f_i(j)$, $s_{ij}^{'} = f_i^{'}(j) \bmod q$ for $j = 1, \ldots, n$ and sends s_{ij}, $s_{ij}^{'}$ to player P_j.

   (b) Each player P_j verifies the shares he received from the other players. For each $i = 1, \ldots, n$, P_j checks if

$$g^{s_{ij}} \hbar^{s_{ij}^{'}} = \prod_{k=0}^{t} (C_{ik})^{j^k} modp \tag{2}$$

   If the check fails for an index i, P_j broadcasts a *complaint* against P_i.

Proceedings of the Third IEEE International Symposium on Network Computing and Applications (NCA'04)
0-7695-2242-4/04 $ 20.00 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on May 13,2010 at 19:21:39 UTC from IEEE Xplore. Restrictions apply.

(c) Each player P_i who, as a dealer, received a complaint from player P_j broadcasts the values s_{ij}, s'_{ij} that satisfy Eq. 2.

(d) Each player marks as $disqualified$ any player that either

i. received more than t complaints in Step 1(b), or

ii. answered to a complaint in Step 1 (c) with values that falsify Eq. 2.

2. Each player then builds the set of non-disqualified players $QUAL$. (We show in the analysis that all honest players build the same set $QUAL$ and hence, for simplicity, we denote it with a unique global name.)

3. The distributed secret value x is not explicitly computed by any party, but it equals $x = \sum_{i \in QUAL} z_i \bmod q$. Each player P_i sets his share of the secret as $x_i = \sum_{j \in QUAL} s_{ji} \bmod q$ and the value $x'_i = \sum_{j \in QUAL} s'_{ji} \bmod q$.

Extracting $y = g^x$ mod p:

4. Each player $i \in QUAL$ exposes $y_i = g^{z_i} \bmod p$ via Feldman VSS:

(a) Each player P_i, $i \in QUAL$, broadcasts $A_{ik} = g^{a_i k} \bmod p$ for $k = 0, \ldots, t$.

(b) Each player P_j verifies the values broadcast by the other players in $QUAL$, namely, for each $i \in QUAL$, P_j checks if

$$g^{s_{ij}} = \prod_{k=0}^{t} (A_{ik})^{j^k} modp \qquad (3)$$

If the check fails for an index i, P_j *complains* against P_i by broadcasting the values s_{ij}, s'_{ij} that satisfy Eq. 2 but do not satisfy Eq. 3.

(c) For player P_i who receives at least one valid complaint, $i.e.$ values which satisfy Eq. 2 and not Eq. 3, the other players run the reconstruction phase of Pedersen-VSS to compute z_i, $f_i(z)$, A_{ik} for $k = 0, \ldots, t$ in the clear. For all players in $QUAL$, set $y_i = A_{i0} = g^{z_i} \bmod p$. Compute $y = \prod_{i \in QUAL} y_i \bmod p$.

2 An algorithm for computing a generator of G_q

The DKG algorithm requires all players to have the same values of p, q, g and h numbers, where p is a large prime number and q is a large prime divisor of $p - 1$. The number g is an element of the multiplicative group Z_p^* of order q and h is an element of the subgroup G_q in Z_p^* generated by g. To find a large prime number we used the Matlab package. After finding the prime number p, we used Matlab to find prime factors of $p - 1$ and took the largest prime from the solution set.

The order of an element in a multiplicative group Z_p^* is defined as the minimum degree to which that number needs to be raised to get the identity element of the multiplicative group. In other words g needs to satisfy the following condition: g^q mod p = 1. The problem of finding g comes down to finding roots of the polynomial x^q - 1 in $mod\ p$ arithmetic. Note that the identity element of Z_p^* is a trivial solution of this equation and will not be taken into account. We have used two approaches to solve this problem. The first approach was exhaustive search, starting from the smallest element of Z_p^* each number is tested and if it does not satisfy the condition x^q mod p = 1 we would proceed to the next smallest element of Z_p^*. However, this approach is not efficient. We have implemented another algorithm based on the bisection method. This new algorithm is supposed to narrow down the interval in which the number g can be found with higher probability and perform the exhaustive search for g on this interval first. The modification to the bisection method was made so that it was possible to be used in modular arithmetics: all elements of Z_p^* less than $p/2$ are considered to be positive and all others are negative. The algorithm takes several input parameters: the primes p and q, maximum number of iterations and the minimal interval size. In each iteration we find the middle point of the interval c, check if is the solution and if it is not proceed recursively on one of the interval halves. If the value of the he function $f(x) = (x^q - 1) \bmod p$ changes the sign when x grows from c to the end point of interval, the upper half is chosen. Otherwise, we set the interval for the next iteration to the lower half. The base interval is set from 2 to $p - 1$. The bisection

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on May 13,2010 at 19:21:39 UTC from IEEE Xplore. Restrictions apply.

is performed a specified maximum number of times or until the length of the interval becomes less then the minimum size required by the algorithm. In every iteration we store the bounds of the intervals that will not be considered. After narrowing down the interval to certain size exhaustive search is performed. If the g is not found in the current interval we proceed with the exhaustive search on the last saved interval. We traverse the intervals in the reverse order until the g is found. In each step the size of the interval is twice the size of the interval in the previous step.

The Algorithm:

1. Set the bounds of the initial interval $a = 2$, $b = p - 1$; set number of iterations n_{max} and minimum interval size $minintsize$; set the bound for positive numbers $bound = p/2$. Calculate the value of the function $f(x) = (x^q - 1) \text{mod}\ p$ at the ends of the interval: $u = f(a)$ and $v = f(b)$ and if $u = 0$ or $v = 0$ display the g and exit.

2. Perform steps 3 to 5 n_{max} number of times.

3. Calculate a new interval bound $c = (a + b)/2$ and the function value $w = f(c)$. If $w = 0$ exit.

4. If w and v are not of the same sign, save bounds of interval from a to c and set the lower interval bound equal to c; else save bounds of interval from c to b and set upper bound equal to c.

5. If $b - a < minintsize$ proceed to $Step - 6$.

6. Perform exhaustive search on interval a to b and exit if g is found.

7. Repeat $Steps - 8$ to 10 until g is found.

8. Set a to the lower bound of the smallest interval that has not yet been searched.

9. Set b equal to the upper bound of the same interval.

10. Perform exhaustive search on interval a to b and exit if g is found.

We have used both algorithms to find the group generator for large primes. The execution times for both algorithms varies highly based on the values of prime numbers p and q, even if the p numbers are roughly of the same size. In all cases the bisection method is either as fast as the exhaustive search or much faster. The significant speedup was obtained when the execution time is long.

The fourth number required, h is an element of the subgroup G_q in Z_p^* generated by g. It is important that nobody knows the discrete log of h with respect to g. To generate h first we find a random number r in Z_p^*. To generate a random element in the subgroup generated by g it is enough to set $h = r^k \bmod p$ where $k = (p - 1)/q$. If q^2 does not divide $p - 1$ then h is in the group generated by g.

3 Implementation of the GJKR-DKG protocol

In this implementation all the players have to run the same code and exchange information without making use of an intermediate server. We followed a client/server model using secure sockets as presented in [7]. This way the player is both able to establish a connection to a peer as well as handling incoming requests.

This has been possible by using threads, which operate independently (our implementation in particular used the pthreads). Our development has been performed on the Intel/Linux platform. We use the RedHat 8 distribution and the GNU C compiler (gcc).

The DKG implementation with 2-players:

The main program spawns a reader and a writer thread. These two threads are in charge of reading and writing the information from/to a shared buffer that is used to keep the information exchanged between the players. While listening for a connection both players attempt a connection to a peer. When both peers have established a connection the main program of each player starts to produce the DKG information. This information is written to the shared buffer, which is monitored by the reader thread. When the reader thread finds a message it fetches it from the buffer and send it to the other player over the OpenSSL connection previously established. On the other side the writer thread is monitoring the connection and receives the message that is then put into the buffer.

This producer/consumer process has been performed using a basic semaphore technique.

The buffer has a semaphore variable that is used to lock the data when something is using it. Both the main program and the threads wait until the semaphore is green (lock=0) before using the buffer. This prevents corruption in the buffer and preserves the order of the operations. When a thread needs the buffer it waits until the semaphore is green (lock=0). When this

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on May 13,2010 at 19:21:39 UTC from IEEE Xplore. Restrictions apply.

happens it sets the lock to 1 (red signal), performs the necessary operations to the buffer, and then releases the lock (green signal).

In the 3-players version the approach is similar but this time we have more outgoing and incoming connections (2+2 per player). The 3-players program has the following changes: (i) When it starts the reader thread establishes a connection to each peer (2). (ii) It listens for incoming connections and it spawns one thread per connection (2).

Figure 1 describes a 3-players run and how data is pulled/written from/to the shared buffer and then exchanged between peers through network connections. In order to simplify the data exchange we designed a specific protocol. When a player sends an object to one or more peers it creates a packet using the following format:

Destination	Sender	Object	Values list

Destination: it is either "B" (broadcast) or the peer id of the receiver. Broadcast sends the packet to all peers.
Sender: id of the sender player **Object**: the object type sent. Types are qualified using the characters **C, S, $, A, !, X**.
'C': C values as defined in DKG algorithm; **'S'**: S share values; **'$'**: S' share values; **'A'**: A as defined in DKG; **'!'**: Complaint sent by a player to all peers; **'X'**: Disqualification (when received used to remove a peer from QUAL set).
Value(s): value(s) of the object
Lock/Unlock The main program and the threads have to compete for the buffer access. Once the buffer is obtained it has to be locked in order to prevent another method either to overwrite or read wrong data. In order to guarantee a safe access to the buffer we used the following method:

While Lock==1 do wait() // wait for buffer to be released

Lock=1; // lock buffer

$<$ modify or read Buffer $>$

Lock=0; // release buffer

Now that we have the 3-players version, which spawns a thread at each incoming connection, it is not difficult to extend this to n players. Then the reader right must open n-1 connection to the different

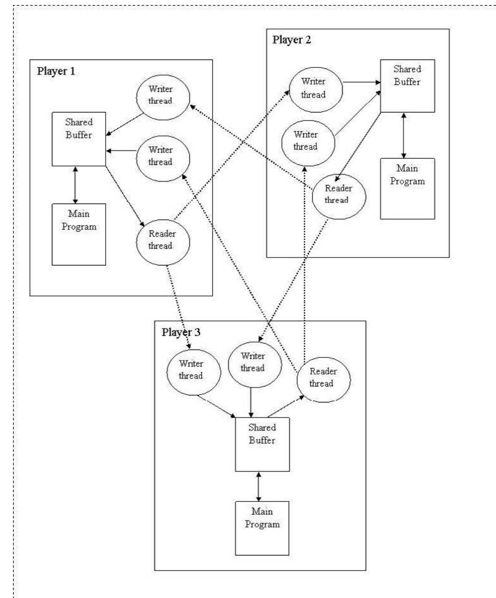

Figure 1. Asynchronous communication between three players using threads

peers. These connections are persistent because they are opened at the beginning of the program and closed at the end.

4 Test Results and Conclusions

Our testing has been performed on a 100Mbps network of Linux workstations with Pentium 4 2.4GHz and 512MB of RAM. We implemented: (I) the Bisection and Exhaustive search algorithms (Matlab); (II) the GJKR DKG algorithm (gcc and OpenSSL ([7]) with two, four and eight players).

Our results are the following: Table 1 contains the execution time of the Bisection and Exhaustive search algorithms to compute g after p and q have been computed.

Tables 2-4 contain the run times (in seconds) of the GJKR DKG protocol, where:

C = total computation time (excluding communication and wait times); R = total read communication time ; W = total write communication time ; TE = total execution time to run the Gennaro protocol;

Conlcusions: (1) Table 1 shows that the Bisection is an efficient algorithm to compute the parameter g which is necessary to run the DKG protocol. (2) Ta-

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on May 13,2010 at 19:21:39 UTC from IEEE Xplore. Restrictions apply.

bles 2-4 demonstrate that the GJKR DKG can be efficiently implemented using secure sockets. We observe that the Read and Waiting times (imposed for synchronization) take the longest time. This is the first attempt (known to us) to implement with the GJKR DKG protocol using secure sockets. In future work, we plan to to optimize our code so that the overall time takes fractions of a second.

Table 1. Execution times for computing the generator of G_q

No. of Digits	p	q	*Bisection*	*Exhaustive*
8	59481613	1652267	1.627e-03	4.685e-03
8	24937123	1873	69.517e-03	92.505e-03
8	14113739	8273	29.498e-03	32.275e-03
8	55607219	7919	78.025e-03	30.327e-03
8	60570721	2003	714.072e-03	697.401e-03
16	2085722970559367	267597923	166	470
16	9307920553388321	245915479	358	1993
16	6466429375385897	1216300843	5	456
16	8374675958515991	2531344841	62	87
32	1851097327260060 8981219760093649	397572450012899 6774316958783	0	0
32	9936189019464045 5210548527110309	178708435601871 322321130444443	0	0.06e-2
32	5014511783579849 5990372909362061	706701248873233 7230134041	47	>900
32	8716662453271704 2911333128095371	8530943806601979 2038651681	8	34
32	5530029326228636 2344764909334001	9216715543714393 72412748889	0	1
48	791630411241235943396043 655700177332679308152851	158326082248247188679208 731140035466535586163057	0	0
48	574256143563558458718976 746753830538032062222159	195844807163071570397304 667742251735226813339	0	0

Table 2. Runtimes of DKG with 2 players

No. of Digits	$P1$	$P2$
8 C	0.09e-2	0.09e-2
8 R	7	7
8 W	0.04e-2	0.04e-2
8 TE	21	20
16 C	0.12e-2	0.12e-2
16 R	6	7
16 W	0.04e-2	0.03e-2
16 TE	20	21
32 C	0.17e-2	0.17e-2
32 R	7	6
32 W	0.04e-2	0.04e-2
32 TE	21	20
48 C	0.24e-2	0.24e-2
48 R	7	6
48 W	0.03e-2	0.04e-2
48 TE	21	20

Table 3. Runtimes of DKG with 4 players

No. of Digits	$P1$	$P2$	$P3$	$P4$
8 C	0.27e-2	0.27e-2	0.33e-1	0.27e-2
8 R	22	15	12	18
8 W	0.06e-2	0.05e-2	0.06e-2	0.46e-0
8 TE	26	25	25	25
16 C	0.33e-2	0.34e-2	0.40e-2	0.34e-2
16 R	21	17	11	17
16 W	0.05e-2	0.05e-2	0.06e-2	0.04e-2
16 TE	27	25	25	24
32 C	0.44e-2	0.44e-2	0.52e-2	0.453e-2
32 R	13	18	13	17
32 W	0.05e-2	0.06e-2	0.05e-2	0.06e-2
32 TE	26	24	25	26
48 C	0.59e-2	0.60e-2	0.72e-2	0.60e-2
48 R	22	15	12	16
48 W	0.07e-2	0.05e-2	0.05e-2	0.04e-2
48 TE	26	25	25	25

Table 4. Runtimes of DKG with 8 players

No. of Digits	$P1$	$P2$	$P3$	$P4$	$P5$	$P6$	$P7$	$P8$
8 C	0.86e-2	0.88e-2	0.1e-1	0.89e-2	0.91e-2	0.89e-2	0.90e-2	0.90e-2
8 R	16	22	11	13	14	15	12	24
8 W	0.10e-2	0.09e-2	0.08e-2	0.08e-2	0.09e-2	0.11e-2	0.11e-2	0.10e-2
8 TE	34	34	34	33	36	35	36	36
16 C	0.10e-1	0.10e-1	0.12e-1	0.10e-1	0.10e-1	0.10e-1	0.10e-1	0.10e-1
16 R	24	25	16	19	17	25	20	17
16 W	0.11e-2	0.10e-2	0.11e-2	0.09e-2	0.08e-2	0.09e-2	0.11e-2	0.11e-2
16 TE	36	34	34	33	34	36	36	37
32 C	0.12e-1	0.13e-1	0.15e-1	0.13e-1	0.13e-1	0.13e-1	0.13e-1	0.13e-1
32 R	20	22	18	18	20	18	18	15
32 W	0.10e-2	0.10e-2	0.11e-2	0.09e-2	0.08e-2	0.09e-2	0.12e-2	0.11e-2
32 TE	36	34	34	33	34	36	36	37
48 C	0.16e-1	0.16e-1	0.19e-1	0.16e-1	0.16e-1	0.16e-1	0.16e-1	0.16e-1
48 R	16	17	19	12	20	12	20	17
48 W	0.09e-2	0.09e-2	0.09e-2	0.07e-2	0.11e-2	0.09e-2	0.12e-2	0.11e-2
48 TE	34	33	34	33	35	35	36	36

References

[1] T. ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Trans. Info. Theory*, IT 31:469-472, 1985.

[2] P. Feldman. A Practical Scheme for Non-Interactive Verifi - able Secret Sharing. In *Proc. 28th FOCS*, pages 427-437, 1987.

[3] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust threshold DSS signatures. In *Advances in Cryptology - Eurocrypt '96*, pages 354-371. LNCS No. 1070, 1996.

[4] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure Distributed Key Generation for Discrete-Log Based Cryptosystems. Eurocrypt '99, pages 295-310, LNCS No. 1592, 1999.

[5] T. Pedersen. A Threshold Cryptosystem without a Trusted Party. In *Advances in Cryptology - Eurocrypt '91*, pages 522-526. LNCS No. 547, 1991.

[6] T. Pedersen. Non-interactive and Information-theoretic Secure Verifi able Secret Sharing. In *Advances in Cryptology - Crypto '91*, pages 129-140. LNCS No. 576, 1991.

[7] J. Viega, M. Messier, and P. Chandra. Network Security with OpenSSL. *O'Reilly and Associates, Inc.* ,2002.

Proceedings of the Third IEEE International Symposium on Network Computing and Applications (NCA'04)
0-7695-2242-4/04 $ 20.00 IEEE

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on May 13,2010 at 19:21:39 UTC from IEEE Xplore. Restrictions apply.